Virtual Machine Support for Aspect-Oriented Programming Languages

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte

Dissertation

von
Diplom-Ingenieur

Michael Haupt
geboren in Olpe

Referentin: Prof. Dr.-Ing. Mira Mezini
Korreferent: Prof. dr. Theo D'Hondt, Vrije Universiteit Brussel

Tag der Einreichung: 27. Oktober 2005
Tag der mündlichen Prüfung: 16. Dezember 2005

Darmstadt
D17

Abstract

Aspect-oriented programming (AOP) is, like structured, logic, functional or object-oriented programming, a programming paradigm in its own right. Each paradigm offers its users certain possibilities to decompose application domains to modules.

The decomposition of complex systems, when adopting the four latter paradigms, frequently leads to the problem that no clear modularisation of certain concerns of the system is possible. This becomes evident, for example, when parts of the functionality of a concern that logically belong together are scattered over several modules of the system and appear to be tangled with these modules' codes. Such concerns that cannot be cleanly modularised are called *crosscutting concerns*.

While the aforementioned problems are met in most of the paradigms when a complex system is decomposed to its modules, AOP regards the problematic crosscutting concerns as modules of their own and introduces a new module concept, called *aspects*.

Apart from state and functionality, an aspect also bundles a description of its crosscutting behaviour. To that end, it describes at which points in the execution of an application its own functionality must be called. These points are called *join points*.

Situations that lead to the occurrence of join points and hence to the invocation of aspect functionality are described using *pointcuts*. A pointcut quantifies over the set of all join points and selects from the set, aided by certain designators, those at which the invocation of aspect functionality is desired. Aspect functionality is represented in so-called *advice* that have the form of procedures or methods.

To let the crosscutting concerns, thus modularised using aspects, take effect in an application, they are "woven into" the application code using special compilers called *weavers*. The locations in application code where advice invocations are woven in are called *join point shadows*. A compiler for an AOP language consists, apart from the usual compiler building blocks, also of a module for evaluating pointcuts, and of a weaver.

The pointcut evaluation module retrieves the join point shadows described by the pointcut and passes them on to the weaver, which weaves aspect functionality into the application. However, it cannot be determined at weave-time for all join point shadows whether join points will indeed occur at them when the application is running. Hence, conditionals are woven in addition at some shadows that decide, based on run-time state, whether advice are to be invoked. These conditionals are called *residues*. In this respect, dynamic pointcuts that depend on the control flow of the application are of special interest: join points may occur only in the context of the execution of a certain method.

Among the different AOP implementations, some approaches have been developed in which weaving does not necessarily occur at compile-time, but at run-time of an application. Weaving at run-time has the advantage that, e. g., in the context of J2EE applications, objects can be dynamically and transparently decorated with certain services that are important in J2EE applications. Moreover, dynamic weaving makes more allowance for the inherently dynamic nature of pointcuts than static weaving—pointcuts quantify over the *execution* of an application.

The different approaches to dynamic weaving however lack the kind of support for

AOP language mechanisms that is common for mechanisms of other paradigms: direct support from the run-time environment. For example, virtual machines for object-oriented programming languages offer direct support for resolving virtual methods via method tables. Such support is at present not available for aspect-oriented mechanisms. Instead, the implementation of a run-time environment for AOP languages is simulated at application level.

This is where this work applies. In the course of this work, *Steamloom* was implemented, a virtual machine with explicit support for AOP mechanisms. Steamloom is an extension of an existing virtual machine for the Java programming language.

Steamloom provides a unified representation of Java bytecodes that is accessed by both the virtual machine and the AOP functionality. The latter utilises the bytecode representation's capabilities to evaluate pointcuts and for weaving.

Moreover, Steamloom supports dynamic weaving by utilising the just-in-time compilers of the virtual machine for dynamically recompiling methods affected by weaving. Apart from that, Steamloom has integrated support for some other AOP mechanisms, of which an efficient evaluation of control flow-dependent residues is especially important.

Steamloom is conceived as a platform on whose basis AOP languages can be implemented experimentally. The pointcut model and weaver are extensible, which allows for the implementation of new elements. This extensibility was used in this work to implement and evaluate three different strategies for control flow-dependent pointcuts.

To measure the value of Steamloom, a comprehensive analysis of different aspects of performance and efficiency of a representative collection of AOP implementations was conducted. It is based on a presentation of implementation details of these AOP implementations that has been achieved on the basis of a unified presentation framework.

Zusammenfassung

Die aspektorientierte Programmierung (AOP) ist, ebenso wie die strukturierte, logische, funktionale oder objektorientierte Programmierung, ein eigenständiges Programmierparadigma. Jedes Paradigma bietet seinen Anwendern bestimmte Möglichkeiten, Anwendungsdomänen in Module zu dekomponieren.

Die Dekomposition komplexer Systeme stößt bei den vier Letztegenannten regelmäßig auf das Problem, dass keine eindeutige Modularisierung bestimmter Belange des Systems möglich ist. Dies äußert sich beispielsweise darin, dass Teilfunktionalitäten eines Belangs, die logisch zusammenhängen, über mehrere Module des Systems zerstreut sind und in diesen Modulen als Einsprengsel im Quelltext auftauchen. Solche nicht sauber modularisierbaren Belange werden als *crosscutting concerns* bezeichnet.

Während sich in den meisten Paradigmen die genannten Probleme bei der Dekomposition eines komplexen Systems in seine Module ergeben, fasst die AOP die problematischen *crosscutting concerns* als eigenständige Module auf und führt ein neues Modulkonzept, die so genannten *Aspekte*, ein.

Ein Aspekt bündelt, zusätzlich zu Daten und Funktionalität, auch eine Beschreibung seines querschneidenden Verhaltens. Zu diesem Zweck beschreibt er, an welchen Punkten während der Ausführung einer Anwendung die ihm eigene Funktionalität aufzurufen ist. Diese Punkte werden *join points* genannt.

Die Beschreibung der Situationen, die zum Auftreten von *join points* und somit zur Ausführung von Aspektfunktionalität führen, geschieht mittels *pointcuts*. Ein *pointcut* quantifiziert über die Menge aller *join points* und selektiert aus ihr mit Hilfe bestimmter Bezeichner diejenigen, an denen der Aufruf von Aspektfunktionen gewünscht ist. Aspektfunktionalität liegt in Form so genannter *advice* vor, welche die Form von Prozeduren oder Methoden haben.

Damit die solcherart mit Aspekten modularisierten *crosscutting concerns* in einer Anwendung zum Einsatz kommen können, werden sie mit Hilfe spezieller Übersetzer, so genannter „Weber“, in die Anwendung „eingewoben“. Die Stellen im Anwendungsquelltext, an denen Aufrufe von *advice* eingewoben werden, heißen *join point*-Schatten. Ein Übersetzer für eine AOP-Sprache besteht neben den bei einem Übersetzen üblichen Teilen auch aus einem Modul zur Auswertung von *pointcuts* und einem Weber.

Das Modul zur *pointcut*-Auswertung findet die von dem *pointcut* beschriebenen *join point*-Schatten und übergibt sie an den Weber, der das Einweben von Aspektfunktionalität durchführt. Jedoch steht zur Webezeit nicht für alle Schatten zweifelsfrei fest, ob an ihnen zur Ausführungszeit tatsächlich *join points* auftreten. Daher werden an manchen Schatten zusätzlich Konditionale eingewoben, die, abhängig vom Laufzeitzustand, darüber entscheiden, ob *advice* auszuführen sind. Diese Konditionale werden *residues* genannt. Von besonderem Interesse sind hierbei dynamische *pointcuts*, die vom Kontrollfluss der Anwendung abhängen: *join points* treten hierbei unter Umständen nur im Kontext der Ausführung einer bestimmten Methode auf.

Unter den verschiedenen Implementierungen von AOP haben sich auch einige Ansätze entwickelt, in welchen das Weben nicht notwendig zur Übersetzungszeit, sondern zur Laufzeit einer Anwendung geschieht. Ein Weben zur Laufzeit bietet den Vorteil, dass,

beispielsweise im Kontext von J2EE-Anwendungen, Objekte dynamisch und transparent mit bestimmten Diensten ausgestattet werden können, die im Rahmen einer J2EE-Umgebung von großer Wichtigkeit sind. Weiterhin trägt dynamisches Weben der inhärent dynamischen Natur von *pointcuts* – sie quantifizieren über die *Ausführung* einer Anwendung – eher Rechnung als ein statischer Ansatz.

Die unterschiedlichen bestehenden Ansätze zum dynamischen Weben lassen jedoch die Art von Unterstützung for AOP-Sprachmechanismen vermissen, die für Sprachmechanismen anderer Paradigmen gang und gäbe ist: eine direkte Unterstützung durch die Laufzeitumgebung. Beispielsweise bieten virtuelle Maschinen für objektorientierte Programmiersprachen direkte Unterstützung für die Auflösung virtueller Methoden mittels Methodentabellen. Eine solche Unterstützung ist für aspektorientierte Mechanismen derzeit nicht verfügbar. Statt dessen wird auf Anwendungsebene die Implementierung einer Laufzeitumgebung für AOP-Sprachen simuliert.

An dieser Stelle setzt die vorliegende Arbeit an. Im Rahmen der Arbeit wurde *Steamloom* implementiert, eine virtuelle Maschine mit expliziter Unterstützung für AOP-Mechanismen. Steamloom ist eine Erweiterung einer bestehenden virtuellen Maschine für die Programmiersprache Java.

Steamloom bietet eine einheitliche Darstellungsschicht von Java-Bytecodes, auf die sowohl die virtuelle Maschine selbst als auch die AOP-Funktionalität zugreifen. Letztere nutzt die Möglichkeiten der Bytecode-Darstellung sowohl zur Auswertung von *pointcuts* als auch zum Weben.

Desweiteren unterstützt Steamloom dynamisches Weben, indem die in der virtuellen Maschine vorhandenen Laufzeitübersetzer für Java-Bytecodes dazu benutzt werden, von Webevorgängen betroffene Methoden dynamisch neu zu übersetzen. Darüber hinaus bietet Steamloom integrierte Unterstützung für einige andere Mechanismen von AOP-Sprachen an, von denen eine effiziente Auswertung bestimmter kontrollflussabhängiger *residues* besonders hervorzuheben ist.

Steamloom ist als Plattform konzipiert, auf deren Basis AOP-Sprachen experimentell implementiert werden können. Sowohl das *pointcut*-Modell als auch der Weber sind erweiterbar, was die Implementierung neuer Elemente erlaubt. Diese Erweiterbarkeit wurde im Rahmen dieser Arbeit genutzt, um drei verschiedene Ansätze für kontrollflussabhängige *pointcuts* zu implementieren und experimentell zu evaluieren.

Um den Wert von Steamloom zu ermessen, wurde eine umfassende Analyse verschiedenster Aspekte der Geschwindigkeit und Effizienz einer repräsentativen Menge bestehender AOP-Implementierungen durchgeführt. Sie basiert auf einer Beschreibung von Implementierungsdetails dieser Implementierungen, die auf der Basis eines einheitlichen Rahmens durchgeführt wurde.

Contents

List of Figures

List of Tables

Listings

1. Introduction

Language mechanisms deserve language implementation effort. This maxim has driven language implementations since the very first programming languages, and it has always had an impact on the design of execution layers for programming languages, at whatever level of abstraction from hardware, and in whatever programming paradigm.

Microprocessor architectures, being the execution layers for machine code, have evolved due to various insights on the mechanisms of assembler code. For example, the realisation that assembly instructions are normally processed in several stages (fetch–process–put) has led to the development of pipelining processors that process each of the three steps in parallel for up to three instructions at a time, effecting significant speedups. Also, multi-level memory cache architectures have been a reaction to the fact that main memory access is expensive. [126]

The evolution of programming paradigms has faced numerous challenges for execution layer design and spawned multiple development efforts for optimal implementations of the mechanisms specific to each paradigm. Structured programming, having been first introduced (though at a very low degree of sophistication) in the invention of procedural abstraction and procedure calls, has led to the invention of method call frames that cleanly encapsulate a procedure's state and help avoid complicated memory management for data local to procedures.

In the other major paradigms, similar developments have taken place to support their mechanisms. For functional programming, run-time environments take care of lazy evaluation. Garbage collector implementations were first introduced in LISP environments. In the logic programming area, efficient implementations of the resolution calculus are used when applying rules to facts to deduce new facts. Object-oriented programming, having been very popular for some time now, has spawned sophisticated implementations of the paradigm's core mechanisms, such as virtual method dispatch.

Crosscutting Concerns and the Aspect-Oriented Paradigm

Aspect-orientation [106, 59] is a paradigm in its own right, introducing the notion of *aspects*, a novel module concept that can be used to encapsulate so-called *crosscutting concerns* in software.

Software is designed and built to satisfy the requirements imposed on it. During the design phase, software systems are typically decomposed into modules that each contribute to the overall system functionality in their own way. In the object-oriented paradigm, such modules are *classes* that encapsulate data and functionality specific to it. The modules, i. e., the classes, represent the *concerns* of the software. Concerns can be constituted of several collaborating classes, but it is also possible to regard a single

class as the modularisation of a single concern. For example, a `String` class represents the concern "representation and basic manipulation of character strings".

Ideally, the decomposition of a system into classes yields a clear mapping of concerns to classes. Unfortunately, this is illusional for complex systems because concerns tend to "cross-cut" each other. In terms of object-oriented decomposition, this means that the implementations of most concerns are clearly mappable to modules (i. e., classes) and collaborations thereof. However, the implementations of certain other concerns frequently end up being *scattered* over modules of the aforementioned kind, and *tangled* with their code.

The clearly mappable concerns follow the so-called "dominant" decomposition [151]. Concerns that cannot be clearly mapped to modules once the decomposition has been established are crosscutting. The decomposition might be adapted to provide a better modularisation for them, but this inevitably leads to other concerns being crosscutting afterwards that were previously clearly modularised.

Object-orientation was used to give an example, but this problem cannot be solved in any of the previously mentioned paradigms. In a sufficiently complex system, there is always a dominant decomposition that makes modularising certain concerns using the employed paradigm's mechanisms infeasible. This circumstance is referred to as the "tyranny of the dominant decomposition" [151].

The aspect-oriented programming (AOP) paradigm introduces a new module concept that allows for cleanly encapsulating crosscutting concerns. Such modules are called *aspects*. Of the several flavours of the paradigm that have been investigated and cataloged by Masuhara and Kiczales [112], the focus in this work is on the *pointcut and advice* (PA) flavour.

AOP languages of the PA flavour are based on the following principles [105]. Crosscutting behaviour, encapsulated in aspects, is regarded as functionality that is to be executed whenever the application it cuts across reaches certain points in its execution. These points in the execution graph of an application are called *join points* (such as method calls, field accesses, etc.). They are quantified over by means of so-called *pointcuts*, which thus are queries over the execution of a program. Whenever a pointcut matches, crosscutting behaviour, represented in the form of *advice*, which are method-like constructs, can be executed. So, in a nutshell, pointcut-and-advice AOP deals with making crosscutting explicit by quantifying over program execution and implicitly invoking functionality when a point from the quantified set of points is reached [62, 107].

The idea of quantifying over applications and implicitly invoking additional functionality has been around since the advent of *event-condition-action* (ECA) rules in databases [47, 125]. Of course, the shape of applications, and therefore the entities over which quantification is done are different in databases than in an "ordinary" programming language. In databases, quantification is facilitated by rich event algebras that allow for expressing complex circumstances at which actions should take place.

The programming models met in object-oriented databases closely resemble those of object-oriented programming languages in that they also comprise classes, objects and methods, and so forth. In object-oriented active databases, where ECA rules are themselves treated as first-class elements of the object-oriented programming model [47]

ECA rules and aspects come even closer [42].

While the relations at the technical level are quite obvious—both ECA rules and aspects react to specific conditions by executing additional behaviour—, the conceptual differences are considerable. Aspects form a *module* concept, bearing dedicated language and compiler support for type safety and so forth. ECA rules, on the contrary, lack the explicit modular nature.

Aspect-Oriented Programming Implementations

Implementations[1] of AOP languages of the PA kind are in the focus of this work. Since the advent of AspectJ [105, 19], an aspect-oriented extension of Java, numerous languages have been developed and implemented. Their implementations all have in common two important cornerstones: *join point shadow retrieval* and *weaving*. The former maps dynamic join points to their corresponding static shadows [113]: code structures (expressions, statements or blocks) that *might* yield dynamic join points during execution. A method execution join point's shadow is a method body, the shadow of a method call is a call instruction, etc. Given a pointcut, the retrieval logic calculates the shadows of join points matched by the pointcut. The shadows are passed to the weaver, which weaves code for dispatching to aspect functionality at these shadows.

For pointcuts that quantify only over static properties of join points, and can thus be directly mapped to code, the dispatching logic is a direct call to advice functionality. However, pointcuts that quantify over dynamic properties of join points in AspectJ cannot definitely be mapped to places in code. This covers, for example, pointcuts that use `cflow`, `target`, `this`, or `args` pointcut designators. The `cflow` designator quantifies over control flows, and `target`, `this` and `args` can be used to filter objects from a join point's context by type, where the latter applies to, e. g., method parameters. For such pointcuts, the dispatching logic also includes pieces of conditional logic (called *residues*) to check for the dynamic properties. Depending on the kind of dynamic pointcuts, the implementation of the residues can be more or less complex.

For `target`, `this`, and `args`, residues can simply be implemented as dynamic type checks. Residual logic gets more complicated for `cflow`; in this case, the application's execution and its entering and leaving control flows need to be monitored. Recent advances in the development of pointcut languages [123], however, go much further with regard to dynamic properties of join points that pointcuts can refer to, taking into account the *history* of application execution, or the dynamic object store. While such models increase the power of pointcuts as referencing mechanism, improving information hiding, dynamic residual logic gets also more complicated [123]. Finally, there are AOP implementations that support "dynamic weaving" [21, 93, 130]: in these systems, it is possible to weave/unweave aspects into/from a running application. Under such circumstances, the set of join point shadows can seldom be determined statically. Due to this, the aforementioned systems insert additional residues at any potential join point shadow.

[1] Some of the material appearing in the following paragraphs has previously been published [79].

The Case for Dynamic Weaving

Being able to perform aspect weaving on a running application is only one argument in favour of dynamic weaving. Actually, it is not very strong when it is not backed by application scenarios.

An interesting application of dynamic weaving is the *transparent* extension of classes and/or their instances with services such as transactions or persistence. Such services are, e. g., used in J2EE containers, where classes have to adhere to very strict protocols to be deployable in an application. The Spring framework [98, 140] aims at reducing the implementation and maintenance overhead as well as the complexity of J2EE container implementations themselves.

In Spring, a J2EE object is simply a JavaBean [91] with no obvious commitments to any J2EE framework. What services are to be applied to such an entity, and in which ways, is specified by providing appropriate configuration data, and the aforementioned services are applied using AOP and dynamic weaving.

J2EE applications, or, more generally, middleware, is not the only application area of dynamic weaving. Scenarios where objects dynamically adopt and abandon *roles* are also conceivable. Objects can be seen from different perspectives during their life-time, and they may be decorated with new facets while keeping their identity [114].

From the perspective of the implementation of aspect-oriented programming language, there is a clear demand for dynamic weaving. As mentioned above, pointcuts are queries over a program's *execution*. This implies that they are dynamic by nature: regarding them as mere queries over a program's *structure* is inappropriate.

Taking this for granted, one must admit that they deserve dedicated *dynamic* support. Sets of dynamic join points resulting from the evaluation of dynamic pointcuts depend on each other, as in the case of `cflow`. These dependencies can, to a certain degree, be determined statically, but this comes at a considerable cost. The dependencies can also be expressed using residues that check for dynamic conditions, but this leads to run-time overheads. A dynamic pointcut may even depend on information that is altogether unavailable for static analysis, such as—possibly even transitive—relationships among objects [123]. A natural approach to dealing with sets of join points depending on each other is to weave dynamically [75].

Critique of AOP Implementations

Taking these notes on dynamic weaving and the above brief description of AOP implementations into account, it can be observed that such implementations provoke criticism in some respect.

Currently, dispatching logic, including residues, is inserted into application code at compile- or load-time. Hence, this logic is executed by the VM as part of the application. Language mechanisms are thus implemented at application level, not at language implementation level.

The same holds for the implementations of AOP infrastructure responsible for, e. g., weaving, management of aspect-related data structures, and so forth. These infrastruc-

tures are often extensible class libraries, and the execution environment is urged to spend much time on executing functionality specific to them.

There is a *semantic gap* observable here: AOP mechanisms are *language mechanisms*, but they are implemented as part of the application being executed. This gap has some impact on performance and debugging. Performance is influenced because executing language mechanisms in the execution layer can be much more efficient than "emulating" them at language level. Debugging suffers from the fact that considerable portions of the call stack trace belong to methods from the AOP infrastructure: the actual call stack of the application is obscured by infrastructural methods being listed therein that are of no interest to the programmer.

Run-Time Support for Aspects

Above, it has been motivated that aspect-orientation is a paradigm in its own right, and that the way its language mechanisms are implemented at application level has certain negative influences. Aspect-oriented language mechanisms, just like previous paradigms' language mechanisms, such as late binding, lazy evaluation or unification, deserve to be integrated in the underlying execution environments. Hence, the thesis of this work is:

> *In order to implement specifically aspect-oriented programming language mechanisms flexibly and efficiently, dedicated support for them has to be integrated in the languages' execution layers.*

This implies that the mechanisms specific to languages supporting the aspect-oriented paradigm have to be identified and implemented accordingly. It also implies that appropriate measurements have to be found to evaluate the quality of such implementations.

An execution layer with dedicated support for AOP should fulfil some requirements:

- Low-level mechanisms specific to aspect-oriented programming should be supported *directly* by the execution layer. That is, the layer should come with dedicated data structures and functionality supporting the execution of such mechanisms. The degree of AOP-specific functionality executed as part of the application should be reduced to the absolutely necessary.

- The set of mechanisms supported should not be restricted to those found in a particular AOP language. Instead, the basic mechanisms that are common to AOP languages should be supported. Also, the execution layer should be extensible with regard to such mechanisms, to facilitate the development and evaluation of new implementation approaches.

- Concepts of aspect-oriented programming in the PA flavour should be supported as first-class entities of the execution layer's programming model. Access to the AOP functionality of the layer should be made available such that it is easily targetable by language compilers. That is, an appropriate set of classes is to be provided that model AOP concepts and allow for interfacing with AOP functionality through an API.

- The performance of such mechanisms should be as high as possible. The performance of those parts of applications that do not take advantage of aspects should not be negatively influenced by the presence of AOP support in the execution layer.

To provide a realisation of such an execution layer, *Steamloom* was implemented, a Java virtual machine (JVM) supporting the asked-for capabilities to some extent. Steamloom is a research prototype implementing AOP mechanisms as part of the JVM's functionality. It was implemented on top of an existing JVM, namely IBM's Jikes Research Virtual Machine [97] ("Jikes" for short).

To evaluate Steamloom's capabilities, especially in contrast to other implementations of AOP, a wide range of performance benchmarks and other measurements was applied to Steamloom and several other AOP implementations. The results gained from these measurements strongly underpin the above claim.

Organisation of this Work

Four chapters follow the introduction to complete this dissertation.

In the next chapter, related work is presented. Even though aspect-orientation is a comparatively young paradigm, the PA flavour alone has already spawned a large number of implementations. They can be grouped into families. A typical representative of each of the families is presented according to a uniform framework. These presentations give a broad overview of AOP implementation approaches. The concluding discussion serves the purpose of detailing the above critique of AOP implementations. In the end, the need for dedicated support for aspect-oriented language mechanisms at virtual machine level will have been made clear.

Chapter 3 is dedicated to a detailed presentation of Steamloom. First the choice of Jikes for building Steamloom is motivated, and this JVM and some of its internals are introduced, to make the following presentation of Steamloom's implementation details better understandable. Steamloom itself is presented in a top-down approach, starting out with a coarse overview of its architecture and the way it interacts with Jikes' building blocks. Its programming model and high-level representation of AOP concepts are presented, before technical details on various mechanisms' implementations are given.

An extensive evaluation of Steamloom and other AOP implementations is made in Chapter 4. Steamloom will be contrasted to the AOP systems presented in Ch. 2. For the discussion, a number of criteria addressing performance and implementation details are applied.

Chapter 5 summarises the contributions of this work, discusses limitations and gives an overview of future work directions.

Acknowledgements

This work would not have been possible without the—sometimes unknowing—help of many people, whom I wish to devote a few lines to.

First and foremost, I am grateful to my thesis advisor, Mira Mezini, who has supported me in many ways for more than five years. My co-advisor Theo D'Hondt has given me various comments and suggestions, for which I am very thankful.

The members of the glorious Steam Team, who have contributed to the implementation of Steamloom, are Christoph Bockisch, Tom Dinkelaker, Michael Krebs, Kai Stroh, Sebastian Eifert, and Sebastian Kanthak. I thank all of them for their unfailing support and enthusiasm that have made this possible. Special thanks go to Christopg Bockisch, without whom this would not have been done. I also thank several people who have done some great work that ultimately led to the inception of Steamloom: Swen Aussmann, Jens Danker, and Marc Eckart.

Jan Vitek, Robert Hirschfeld, Stefan Hanenberg, and Matt Arnold have been of great help in clarifying my thoughts on the subjects of this work. Marc Ségura-Devillechaise, Michael Engel, Bernd Freisleben, Stephan Herrmann, Éric Tanter, and Wim Vanderperren have supported me during writing this dissertation, for example by abiding my insisting questions on details of some AOP implementations, or by just encouraging me. I very much appreciate their help.

The AOSD-Europe Network of Excellence (European Union grant no. FP6-2003-IST-2-004349) has funded part of this work. While I am thankful for this, I am even more happy to have become acquainted with some very nice people in the course of working on the project. They too have had a share in this work by supporting or encouraging me. My cordial thanks and greetings go to Johan Brichau, Ruzanna Chitchyan, Alessandro Garcia, Jacques Noyé, Awais Rashid, and—last but *definitely* not least—Paula Robinson.

Some of my colleagues at the Software Technology, Aspect-Oriented Programming and Computer Architecture Groups of Darmstadt University of Technology have given me support over the past years—each in their own way, be it by means of music, jokes, anecdotes, technical hints, or just by reminding me of the fact that the world is not *that* bad after all. I thank Ivica Aračić, Jörg Baumgart, Michael Eichberg, Vaidas Gasiunas, Wolfgang Heenes, Sven Kloppenburg, Klaus Ostermann, Shadi Rifai, and Thorsten Schäfer.

I cannot possibly make it up to Gudrun Jörs, who has, albeit being busy most of the time with managing the affairs of two to three university research groups, still always had some time left for a chat, or just for listening. I am very grateful for her kind support.

There are some people that have helped me very much, even though they may not know it. I thank my good old friends Stephan Austermühle, Thomas Bartzick, Martin Jacob, Christoph Sasse, Christian and Elisabeth Schmittmann, and Jochen and Nicole Siegbert. I also especially thank John Hunt for books, whiskys and many nice chats.

I am deeply indebted to my parents, who have continuously encouraged me over so many years.

This work is dedicated to my beloved wife, Natascha, who has tolerated many all-night working sessions and week-ends with admirable patience.

2. AOP Implementations

2.1. Introduction

This[1] chapter is dedicated to the presentation of recent project outcomes in AOP research and development. In their presentation the various systems' respective programming models will be described insofar as it is required for a thorough understanding of their execution models. The latter, being of utmost interest for this work, will be presented in more detail.

The systems presented here range from research projects to industrial-strength implementations. For both the programming and execution models, a uniform presentation framework will be used to ensure a levelled description. In designing this framework, existing work aimed at classifying and comparing AOP implementations was taken into account.

The presentation is ordered according to certain characteristics of AOP implementations that allow for grouping them into "families". A detailed evaluating discussion of the systems is done in Ch. 4, together with the discussion of the contributions of this work, to contrast them directly and in one place.

The discussion following the presentations illustrates shortcomings of the presented systems. The result of the discussion is a critique of existing approaches that motivates the need for dedicated AOP support at virtual machine level.

The structure of this chapter is as follows. To begin with, existing approaches to the classification of AOP systems are described. Next, the framework used in the presentations and analyses of the various systems is introduced. After that, the system families along which the presentations are organised will be described. Each of the different families of systems will then be introduced by presenting in detail typical and prominent representatives of each of them. A discussion of the presented systems rounds off this chapter.

2.1.1. Approaches to Analyse and Classify AOP Systems

Masuhara and Kiczales [112] have devised a framework to identify several flavours of aspect-oriented programming. Their framework identifies four major families of AOP mechanisms that each apply specific techniques to support a certain kind of crosscutting: pointcuts and advice, traversal specifications, class composition and open classes. The focus of this work is on implementations of systems supporting the *pointcuts and advice* (PA) flavour of AOP. In this field, few approaches to a structured and comprehensive classification of such AOP implementations have been published so far.

[1] Part of the material in this chapter has previously been published [34].

Chitchyan and Sommerville [41] have compared several implementations of *dynamic* AOP. According to their definition, an AOP system is considered dynamic "if it ... accommodates dynamic change with *crosscutting* concerns". Thus, the AOP systems taken into account are such that they allow for adding and/or removing crosscutting concerns to/from a running application. To the end of analysing and classifying such systems, Chitchyan and Sommerville have set up a framework of their own, that is however naturally restricted to the examined systems' support for dynamic weaving. Also, only Java-based systems have been surveyed. Nevertheless, the framework provides some relevant guidelines for the work presented here.

Part of the work going on in the AOSD-Europe project [10] is dedicated to surveying existing AOP systems. Two reports, one on aspect-oriented middleware [110] and one on programming languages and execution models [34] have been completed that each follow a structured approach in analysing their respective subjects. The two reports cover implementations in far more languages than only Java; the systems range from C++, C# and COBOL over Java to Smalltalk. These documents have also been influential for the design of the framework used here.

Comparing Dynamic AO Systems

Chitchyan and Sommerville's analysis framework [41] is, as mentioned above, focussed on systems supporting dynamic weaving. They however see dynamic AOP implementations as a more generic type of systems with support for *dynamic reconfiguration*. The surveyed systems are classified along generic criteria for systems of the latter kind, and along two additional axes, namely *weaving time* and *weaving technique*.

Weaving *time* is subdivided as follows:

- *Load-time* weaving modifies application classes as they are loaded into the JVM.
- In a system with *JIT compiler* weaving, the modification of application code takes place when the JIT compiler translates it to native code.
- When *dynamic proxies* are used, a specific facility of the Java programming language is exploited to support the reflective invocation of advice.

The classification of the weaving *mechanism* addresses the question of how application code is modified to eventually invoke advice functionality:

- First, it is possible to insert so-called *hooks* into application code that jump to an AOP infrastructure which is responsible for branching execution to advice. An implementation can perform *total* or *actual* hook weaving, inserting hooks at all possible join point shadows, or only at code locations explicitly specified as join point shadows, respectively.
- The third weaving technique is called *collected* weaving; it avoids using hooks at all and instead weaves direct invocations of advice code at join point shadows.

The generic criteria for systems supporting dynamic reconfiguration take into account various aspects of such systems. They address

- the ability of a system to define an application in terms of loosely coupled modules, i. e., whether it is possible to look at a single module and its implementation (functional view) as well as at the entire application as a collection of interconnected modules (structural view),
- the support of a system for the application's integrity, i. e., whether the system can ensure that application state and behaviour are consistent upon reconfiguration,
- the degree of isolation of the application from the system, i. e., in how far the application is or can be involved in the reconfiguration process,
- the way a reconfiguration specification is represented,
- the efficiency of the system, expressed in, e. g., the delay imposed on the application upon reconfiguration, and
- the robustness of *programmed* (i. e., application-internal) changes with respect to *evolutionary* changes (i. e., changes induced from the outside).

As the authors mention, the AOP-specific criteria were derived from existing Java-based systems by observation and generalisation. Nevertheless, this framework indeed covers a wide range of AOP implementations, even some that were not explicitly dealt with in the investigation. Still, the way the framework was created leads to some insufficiencies.

AOP systems that perform weaving at the meta-level without interacting with application code at all, such as AspectS (cf. Sec. 2.10), are not covered by the weaving time criteria. Also, the "dynamic proxies" and "JIT compiler" classes appear to be inappropriately named. Indeed, both the JIT compiler and dynamic proxies operate at run-time, but the two names denote *utilities used to implement weaving* rather than a *time* at which weaving is applied.

The explicit constriction of the framework to *dynamic* systems leads to compiler-based static weaving systems naturally not being covered. This is not a point of criticism, but highlights that the framework devised by Chitchyan and Sommerville is not applicable to the work presented here.

Aspect-Oriented Middleware

The systems in the focus of the AOSD-Europe middleware survey [110] are naturally not of the kind that is of main interest to this work. The report focuses on middleware implementations with a special interest for their support for component models, while the focus of this work is on language implementation. Nevertheless, the report contains a structured approach to comparing aspect-oriented middleware implementations, some of which are actually language implementations with dedicated support for middleware.

In the report, *customisability* and *usefulness* are mentioned as the two most important criteria based on which approaches are evaluated. From these two starting points, the authors have derived several criteria for the comparison of middleware implementations based on AOP, namely:

- the aspect-oriented programming model, which may be an established component programming model, such as CORBA or J2EE, or specifically developed with the particular AOP mechanisms found in an implementation,
- the primary entities (e. g., objects, components, or agents) supported by the system,
- the *static/dynamic* nature of the weaving model, according to the time when weaving occurs (compile-time, deployment time, load- or run-time),
- the *invasive/non-invasive* nature of the weaving model, depending on whether internal properties (such as private methods and fields) of core entities can be affected (invasive), or whether only publicly exposed interfaces can be used for interception by aspects (non-invasive),
- reusability of aspects, which is not regarded across middleware implementations, but across applications deployed in a particular middleware container, and
- extensibility and adaptability of applications developed in the system, which is influenced by the possible degree of dynamic weaving, and by the degree to which aspects can be programmed declaratively.

These criteria are defined at a high level of abstraction and do not exhibit great interest in implementation details. This is natural since the report's goal is to compare aspect-oriented middleware containers with regard to adopters that use them in an environment where they are used rather than analysed at a technical level.

Aspect-Oriented Languages and Execution Models

The focus of the AOSD-Europe report on AOP languages and their implementations [34] is on presenting such systems as a means to *write* aspect-oriented software on the one hand and as a means to *run* aspect-oriented applications on the other. Both languages and execution models are presented along the lines of structured frameworks that have the form of questionnaires. The two questionnaires aim at, by being filled in, describing the particular features of languages and their implementations along several distinct dimensions.

For the languages, the criteria address the language-level representation of aspect-oriented concepts and the way they can interact with base application code. Their expressiveness is evaluated as well as the available means to control their application and interference. The dimensions are:

- the language's join point model and pointcut language,

- its advice model and language,
- the employed aspect module, instantiation and composition models, and
- the weaving model.

The execution model criteria regard the realisation of language-level concepts at the level of the execution layer dedicated to applying them to programs. From an architectural point of view, such an execution layer can be provided in the form of a framework or library, or even as a virtual machine. At the execution layer, the details of the data structures used to implement the various models are of interest, and the workflows employed in performing the various typical tasks to be met in an AOP implementation. The dimensions are split into a *model* and a *functionality* part. The former gathers details on the implementations of the various models that can be found in a language. They are:

- the implementation's architectural characteristics, and
- the implementations of the employed aspect, advice and pointcut models in terms of the data structures used for their representations.

The functionality part's interest is in describing in detail the different processes applied and their results, namely:

- join point shadow retrieval,
- weaving (also covering the structure of woven code, i. e., weaving results),
- the treatment of dynamic pointcuts,
- advice instance management, and
- the workflows of aspect deployment and undeployment.

One important point of interest in this report is that it explicitly looks at the degree to which aspects, pointcuts, advice, and other AOP concepts are integrated in the language. This is done by evaluating whether these concepts are available as first-class entities in the language or at run-time.

This report covers a wide range of AOP implementations based on several programming languages. Not all of the surveyed languages' execution models are covered, but due to the likeness of many language implementations despite of the respective languages' differences, this was not deemed necessary in writing the survey.

The frameworks defined in this report come close to what is necessary to suit the description of AOP system implementations that is to follow below in this work[2]. This holds especially for the execution models questionnaire, which has been of great influence in designing the framework described in Sec. 2.1.2.

[2]The author of this dissertation has co-edited the AOSD-Europe report and designed the execution model description framework used therein.

2.1.2. Presentation Framework

The framework that is used here for the presentation of AOP implementations is designed with a strong regard to execution models, as they are the main focus of this work. Altogether, it is intended to give an overview of the design space of aspect-oriented execution models. Since the AOSD-Europe report on languages and execution models, as described in the previous section, also has a strong emphasis on execution models, that report has had a significant impact on the design of our framework.

Due to its focus not being directly on language implementations as such, the report on aspect-oriented middleware systems has not had direct influence on the design of the framework. Still, some of its aspects are implicitly present; namely those which focus on the AOP language implementation side of aspect-oriented middleware systems. The static/dynamic weaving criterion is an example for this, but it is to be found in the languages and execution models report as well, and is there defined at a greater level of detail.

The framework is formed as a collection of "topics of interest", for each of which a number of questions is formulated. Answers to these questions contribute a comprehensible description of a system in question. Below, the framework's elements are given in the order they are applied to an AOP system. Also, additional explanations of the questions and definitions of terminology are given.

A. Language Presentation The AOP language of the system in question is briefly described. The presentation does not go into deep detail, though; it is restricted to the crucial features that are of interest especially with regard to developing a thorough understanding of the execution model.

At the end of the language presentation, a simple example of an aspect is given to give an impression of the programming model of the language in question. The example uses the simple base application shown in Lst. 2.1. The sample aspect to be applied to this base application is also very simple: it attaches a before advice to the call (or execution, depending on the support of the particular AOP system for join points) of the `hello()` method.

```
1 public class HelloWorld {
2   public static void main(String[] args) {
3     HelloWorld h = new HelloWorld();
4     h.hello();
5   }
6   public void hello() {
7     System.out.println("Hello, world!");
8   }
9 }
```

Listing 2.1: Base application for the AOP language examples.

B. Execution Model Architecture The implementation of the execution model is described from an architectural point of view. This includes the basic characteristics of

the applied mechanisms, but abstracts away from the implementation details.

B.1 How is the model implemented, relatively to the execution model of the language this AOP system extends? This question directly addresses the architecture of the AOP system. For example, in the Java domain, a system can be implemented by means of a modified class loader, or as a VM plugin. The question also reflects on the way access to the AOP functionality is provided, e. g., through a library with an API, or a framework.

B.2 At what stage of an application's life cycle are AOP mechanisms applied? The distinction to be applied in this case is that between compile-time, load-time, and run-time. Strictly speaking, this distinction does not cover approaches that postprocess native code or bytecode. These are subsumed under the term compile-time as well, since they operate, like compile-time approaches, before any execution environment gets involved. This question also reflects on the type of weaving in terms of being static or dynamic. To distinguish static from dynamic weaving, the definition of Chitchyan and Sommerville as described above [41] is applied.

B.3 What are the basic techniques employed to enable AOP mechanisms? The question addresses the mechanisms used to enable aspect interaction with base application code. An AOP implementation could, for example, use meta-level composition, hook insertion, or dynamic recompilation of methods.

C. Programming Model Implementation In this section of the framework, details about the data structures representing the programming model entities, namely aspects, advice, pointcuts and join points are dealt with.

C.1 How are aspects, advice, pointcuts and join points modelled and represented internally? The details of the data structures and their interconnections are of interest here.

C.2 How strong is the support for aspects as first-class entities? The degree of possible dynamism expressible in an AOP implementation increases with the growing ability to represent aspects and their related concepts as first-class entities. For instance, a complete first-class representation of all of these concepts easily allows for dynamically assembling aspects as they are needed.

C.3 Do aspects and advice have to adhere to some protocol? It is common in AOP implementations that aspects have to extend certain classes, or that methods used as advice must have a specific signature. This issue is addressed by this question.

D. Join Point Model Implementation This section aims at highlighting how the types of join points that the join point model of a particular AOP implementation defines are exposed and made available for further processing.

D.1 What model(s) of the application is (are) used to expose join points? The most simple model to be used is an abstract syntax tree (AST) of the application in question, but other models are imaginable, e. g., the object graph [123].

D.2 In what form is (are) the application model(s) exposed to other AOP functionality? Numerous building blocks of an AOP implementation rely on some form of representation of the application they operate on. This question asks for the interface to the join point

model they can rely on.

E. Pointcut Model Implementation While the preceding section deals with the way application features are exposed, this section is about how the exposed features are exploited. Thereby, this section is the first to explicitly address *functionality* rather than *data structures*.

This section of the presentation framework contains only one question: *How is join point shadow retrieval implemented?* This question addresses the implementation of pointcut evaluation, i. e., how the join point model is exploited to find the set of join point shadows belonging to a given pointcut. As an answer to this question, an overview of the retrieval process is to be given. If there are specific differences between the evaluation processes for statically and dynamically determinable sets of join point shadows, they should be described.

F. Weaving Implementation This section addresses questions around numerous aspects of weaving, including details on the shape of woven code.

F.1 How are classes/methods transformed during weaving? The looks of woven code are of interest here. This question also addresses the data structures representing methods and classes and the way they are gained.

F.2 How are advice invoked? The interest is on the functionality that is triggered at join point shadows and ultimately leads to the execution of advice. For example, they might be invoked directly, or some amount of infrastructural code may be necessary beforehand. A sequence diagram is used to illustrate the control flow associated with executing a simple before advice attached to a method call or execution. The example aspect given in section A of the presentation is used for this.

F.3 How are residues implemented and supported? Non-statically determinable join point shadows frequently encompass the use of residual logic. There are two points of interest here, namely whether there is specific support for residues in the AOP implementation, and how woven code constituting residual logic looks.

G. Advice Instance Management This section addresses the way advice instances are associated with the classes for which they represent aspect state and behaviour.

G.1 How is advice instance management performed in general? All AOP implementations support generally applicable aspects that have generally one instance to which all invocations of advice functionality are sent. This case is dealt with in this question.

G.2 How is advice instance management performed for scoped aspects? Where a particular AOP implementation allows for assigning single advice instances to single objects or threads, this question shall describe how such associations are established. A description of scoping is given in Sec. 3.10.

H. Dynamic Deployment Workflow This section applies only to systems that support dynamic aspect deployment. Most of the mechanisms of interest here should already have been explained in the previous sections. It is the purpose of this one to give an impression

of how the different mechanisms cooperate. The process of dynamic deployment is of most interest, because it encompasses important details concerning the interoperation of the various weaving components. Conversely, undeployment is mostly rather simple as compared to deployment, consisting of a few steps that revert the effects of deployment in a straightforward way. Hence, the descriptions of undeployment workflows are, if at all given, rather brief.

I. Other Systems In this section, an overview of systems belonging to the same family is to be given. These systems are only to be mentioned and very briefly described. Where notable differences to the system described in detail exist, these can also be outlined.

2.1.3. Organisation

Applying the above framework to a particular AOP implementation yields a brief description of the language (including a simple example) preceding the detailed presentation of the execution layer. The presentation is structured according to the sections of the framework, and each of the particular questions posed by the framework is answered. In case additional information about a particular system is to be provided that is of special interest to this work, it is given immediately after the actual presentation.

The AOP implementations presented in the following sections of this chapter are each typical representatives of a certain family of systems. For some of the families, more than one representative are described. This is mainly done when different implementations with interesting characteristics exist that still belong to the same family. The families, in the present order, exhibit an increasing degree of the adoption of meta-level facilities in the expression of aspects and the implementation of AOP functionality. In other words, the degree to which aspects are expressed with language rather than API means decreases over the presentation.

The families are as follows:

1. *Compiler-supported static AOP.* Systems in this family do not support dynamic weaving and are based on a dedicated aspect language that is processed by a compiler. AspectJ 5 [19] is a typical representative.

2. *Compiler-supported dynamic AOP, entirely at application level.* Members of this family also are based on language extensions and depend on a compiler, but they support dynamic weaving. Still, they implement *all* of their dynamic weaving capabilities at application level, i. e., using no special facilities of the underlying execution environment. A representative of this family is CaesarJ [38].

3. *Compiler-supported dynamic AOP with environment support.* The difference of this family to the previous one is that systems belonging to it address the underlying execution environment to support dynamic weaving. Arachne [12] and JAsCo [89] represent this family.

4. *Configuration-driven dynamic AOP with environment support.* A "configuration" consists, e. g., of an XML file, or of Java 5 meta-data annotations. AspectWerkz

[21] is presented as a member of this family. It can be argued that AspectJ 5, since it also supports XML- and annotation-based definition of aspects, belongs to this family as well. However, AspectJ does not exploit the run-time environment and does not support dynamic weaving.

5. *Dynamic AOP framework with environment support.* In this category, none but the standard features of the underlying language are used to express aspects, i. e., a standard compiler can be used. The aspect definitions take place by providing classes extending the framework classes accordingly. Internally, specific mechanisms provided by the run-time environment are used to implement aspect-oriented behaviour. PROSE [130] is a representative of this family.

6. *Meta-programming for AOP.* Systems belonging to this family exploit the meta-programming capabilities of the used programming language to support aspects. There are three subgroups. In the first, a purely introspective model based on structural reflection is used. Spring AOP [141] represents this group. The second subgroup provides a more complete reflective model supporting behavioural reflection. This group is represented by Reflex [133]. Lastly, a fully reflective environment allows for operating solely at the meta-level to implement AOP. AspectS [20] is used as an example for such a system.

2.2. AspectJ

A. Language Presentation

AspectJ [105, 19] is an aspect-oriented extension to Java. Since its inception in 1997 [106] AspectJ has evolved to what is perhaps the most mature AOP language currently existing. It is widely used as a reference to which other languages are compared, and its terminology has influenced the development of a wide understanding of AOP.

AspectJ supports both *static* and *dynamic* crosscutting. Static crosscutting comprises extending the structure of application classes by, e. g., letting them extend new super classes, implement new interfaces. It also allows for providing default implementations for methods defined in interfaces. The static crosscutting support is not subject to this work and hence not covered in the discussion below.

Dynamic crosscutting is about altering an application's behaviour by specifying pointcuts and advice. AspectJ's dynamic join point model is very expressive and allows for the definition of fine-grained interactions of aspects with base applications. The available join points comprises, e. g., method calls and executions (where the difference lies in whether a join point occurs at the call site or in the called method), field read/write accesses, and initialiser executions. The advice model is also very complete in that it supports before and after as well as around advice.

Until version 1.2, AspectJ has been a pure language extension. The upcoming version, AspectJ 5 [43], allows for defining crosscutting structure and behaviour through source code annotations or even in XML configuration files. The description below focuses on AspectJ 5.

The sample aspect for the `HelloWorld` application can, in AspectJ 5, be implemented in the two ways presented in Lsts. 2.2 and 2.3. The former listing shows an annotation-based definition of the aspect that is possible due to the support of Java 5 features in AspectJ. The latter shows the same aspect in standard AspectJ syntax.

```
1 @Aspect
2 public class HelloAspect {
3   @Before("call(void HelloWorld.hello())")
4   public void advice() {
5     System.out.println("I am an advice.");
6   }
7 }
```

Listing 2.2: Hello world aspect in AspectJ using annotation style.

The annotation-style definition of the aspect declares the class in the listing to be an aspect using an annotation. The class itself does not differ from an ordinary Java class; there is no special syntax being used. All AOP-related concepts are expressed using annotations. The ordinary method `advice()` is declared to be a before advice using the `@Before` annotation, which also accepts an expression defining the pointcut to which the advice is to be attached.

```
1 public aspect HelloAspect {
2   before(): call(void HelloWorld.hello()) {
3     System.out.println("I am an advice.");
4   }
5 }
```

Listing 2.3: Hello world aspect in AspectJ using AspectJ syntax.

Standard AspectJ syntax uses special keywords like `aspect`, `before` and `call`.

B. Execution Model Architecture

B.1 AspectJ is written in pure Java. AspectJ primarily consists of a 2-stage pipeline: an extended Java compiler that understands the additional language constructs and a binary weaver.

The compiler is a derivative of the Eclipse JDT compiler, whose parser has been extended to understand the new elements of the AspectJ syntax. Also, its type resolution system was modified to take inter-type declarations into account. The compiler produces Java class files as output—it performs no weaving operations. Aspect constructs are captured in the class files as class file attributes attached to relevant members. For example, a before advice in an aspect is output as a special method in the class file with a Java attribute attached to it that captures the fact that it represents a before advice and what its pointcut was.

The weaver accepts class files, which may have been produced by an arbitrary Java source to bytecode compiler. The weaver understands the attributes attached to as-

pects during compilation and performs the necessary pointcut matching and bytecode modification/weaving using a derivative of BCEL [26, 46] optimised for performance.

Applications built with AspectJ run on *any* standard-compliant JVM. The weaving process however introduces dependencies on AspectJ infrastructure classes. These classes are contained in a single JAR file which must be given on the classpath. Apart from that, there are no prerequisites.

Access to the AOP functionality is provided via the language extensions AspectJ brings with it. There are certain reflective access capabilities on join point contexts that are provided through pseudo variables. These introduce a small API for very restricted purposes.

B.2 AOP mechanisms can be applied at multiple points during an application's lifetime; at compile-time or at load-time. The advantage of compile-time weaving is that AspectJ provides excellent feedback on how the aspects are applying to code, which can be exploited by a GUI such as AJDT [2]. The advantage of load time weaving is increased flexibility. Load-time weaving is done by instructing an existing class loader to delegate to the AspectJ weaving adapter when it loads any bytes. The adapter is given a chance to modify the bytes before they are passed to the JVM to actually define a class. There is no support for run-time weaving.

B.3 The basic weaving mechanism used in AspectJ is the weave-time insertion of invocations of advice directly into base application code, namely at the join point shadow which matched the according pointcut. In some special cases, advice are directly inlined at join point shadows for performance reasons (e. g., in case of around advice; cf. below).

C. Programming Model Implementation

C.1 Aspects are modelled as normal Java classes, and aspect instances are ordinary Java objects. The latters' life cycle is carefully controlled: in user code, it is not possible to create aspect instances; the woven code ensures they are created and destroyed as appropriate.

Advice exist as regular methods in the aspect classes, with an internal name. In the case of around advice, the advice body is inlined at the matching join point, for performance reasons.

Pointcuts are captured as attributes in the class file.

C.2 Aspects and advice exist as first-class entities to the same degree as normal Java classes do. They can be directly referenced, methods can be called and fields can be accessed on them. Their life cycle, however, is beyond direct programmatic control by the user.

The class file representing an aspect looks almost like the class file for a normal class. The difference lies in various attributes being attached to the class and method constructs in the class file. The attributes capture the class's aspect nature and its advice and pointcuts.

AspectJ 5 supports a meta-protocol that allows for run-time querying an aspect to obtain information on its advice, pointcuts, and so forth. This protocol, provided through the standard reflection capabilities of the Java 5 platform, allows for obtaining extensive aspects-related meta-level information contained in a class file representing an aspect.

For example, an aspect can be queried for its advice by invoking `getDeclaredAdvice`, and its pointcuts can be retrieved through `getDeclaredPointcuts`.

C.3 Aspects do not have to adhere to some protocol because they are directly declarable in the AspectJ language.

D. Join Point Model Implementation

D.1 The application model that is used for join point exposure is the application's AST, represented as its bytecode.

D.2 The bytecode is exposed to the AOP functionality—i. e., the weaver—by transforming it to a BCEL [46, 26] representation during weaving. Once weaving has been completed, the bytecode is converted to `byte` array form again.

E. Pointcut Model Implementation

Prior to actual weaving, two kinds of objects are created, namely *type mungers* that modify the type hierarchy for a class or change its set of members, and *shadow mungers* which are responsible for matching pointcuts and implementing dynamic crosscutting behaviour. The following discussion focuses on the latter. A shadow munger [81] contains a pointcut and knows how to transform base application code that matches this pointcut to introduce aspect behaviour.

A visitor is used that visits the class, its methods and the code within the methods. At each stage the visitor creates the relevant kind of shadow for matching. For example, on visiting the class, the shadow for static initialisation is created. As soon as the process reaches the code level, BCEL is used to iterate over the bytecode. During this process, the right kind of shadow is created for each particular instruction.

Shadow matching uses the bytecode form of a class. To optimise performance, a "fast match" process is employed that attempts to determine very quickly whether a shadow could match a pointcut. The return value from fast match is a fuzzy `boolean`, indicating whether a pointcut definitely matches, definitely *not* matches, or whether more analysis is needed.

For example, for a `within` pointcut, the fast match approach can very quickly determine whether a type matches or not. This helps in eliminating as many "unnecessary" checks as possible before continuing with the slow process of analysing the bytecode in detail, creating shadows for every field get/set or method call/execution.

Statically determinable sets of shadows are, on the one hand, those that are retrieved by the fast match approach. On the other hand, those shadows that can fully be said to match a given pointcut by compile-time analysis only are also statically determinable.

For join point shadows that match statically, code for invoking advice is immediately inserted.

Dynamically determinable sets of shadows require, in addition to compile-time analysis, a certain amount of run-time analysis to find whether they match a pointcut. When a shadow cannot be matched by the aforementioned compile-time analyses, so-called *residues* are woven at it. Residues are responsible for performing the dynamic checks, e. g., type checks or control flow matching, that cannot be done at compile-time because the required state is not yet existing.

F. Weaving Implementation

F.1 During weaving, new members can be added to classes for a couple of reasons. For example, infrastructure may be required to support the aspect life cycle. The methods `aspectOf()` and `hasAspect()` are added to aspect-representing classes. They are used to facilitate advice calls.

For example, if an aspect is declared to be instantiated `pertarget` (or alike), a new member field is added to target classes. In each instance of the target class, this field references the advice instance coupled to the target instance.

If the `thisJoinPointStaticPart` pseudo variable is used to access join point context information in any advice, a new field for carrying such information is added to all matched target classes. It is initialised at compile time and used the according advice is called.

Other newly introduced members are required to act as accessors in the case of a *privileged* aspect that is able to access private members of a target class. To achieve this, accessors are generated in the corresponding target class that expose the field to the aspect.

Base application methods are modified to invoke advice at join point shadows. For example, if the aspect shown in Lst. 2.2 above is applied to the `HelloWorld` application, weaving results in code looking like this (there is no source code representing the woven output since weaving is done on bytecode directly, but decompilation would yield a result like this):

```
1 public static void main(String argv[]) {
2   HelloWorld h = new HelloWorld();
3   HelloAspect.aspectOf().advice();
4   h.hello();
5 }
6 ...
```

The static `aspectOf()` returns the instance of the aspect, and the actual advice invocation simply invokes `advice()` on that instance.

Around advice are treated differently. Based on performance considerations, for which an analysis is done during weaving, the advice are directly inlined into the base application code, or closures are generated and the advice invoked on them.

Details on the semantics of the particular invocations, and on around advice, are described in the next few paragraphs.

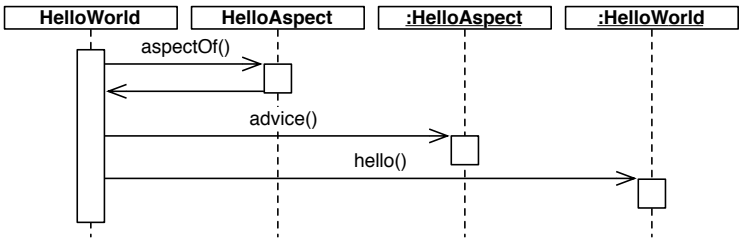

Figure 2.1.: Invocation of a before call advice in AspectJ.

F.2 The `aspectOf()` static method is called on an aspect to obtain the right instance upon which to call the advice. The `aspectOf()` method does not necessarily exist without parameters; when `pertarget` or related constructs are used, an according parameter (i. e., the target of an operation in this case) is passed. In any case, `aspectOf()` returns the advice instance appropriate for the join point shadow at which it was called.

After the advice instance has been retrieved, the advice is immediately called without further ado. No further infrastructure is involved in the actual invocation. The outline of the invocation of the sample advice is shown in Fig. 2.1.

Around advice, as mentioned above, are treated in a special way. There is a dual weaving approach for them: wherever possible, advice code is *inlined* into base application code for performance reasons.

For the example class and aspect in Lst. 2.4, the inlining weaving approach produces code as in Lst. 2.5. The original join point shadow, the call to `X.y()`, is moved to a new method that is called from within the advice. The advice in turn is implemented as a `private static final` method an invocation of which replaces the original join point shadow.

```
1  public class X {
2    public static void main(String[] args) {
3      X x = new X();
4      x.y();
5    }
6    public void y() { System.out.println("y"); }
7  }
8
9  public aspect A {
10   void around(): call(void X.y()) {
11     System.out.println("-->");
12     proceed();
13     System.out.println("<--");
14   }
15 }
```

Listing 2.4: Sample source code for around advice weaving strategy demonstration.

Inlining is impossible, though, when an around advice may apply to itself, or to an instruction within it. In that case, closures are used. The generated code can be seen from Lst. 2.6. The original shadow is again moved to a dedicated method that was

```
1  public class X {
2    public static void main(String[] args) {
3      X x = new X();
4      y_aroundBody1$advice(x, A.aspectOf(), null);
5    }
6    ...
7    private static final void y_aroundBody0(X x) {
8      x.y();
9    }
10   private static final void y_aroundBody1$advice(
11     X _this, AroundClosure ajc_aroundClosure, AroundClosure aroundclosure
12   ) {
13     System.out.println("-->");
14     AroundClosure aroundclosure1 = aroundclosure;
15     y_aroundBody0(_this);
16     System.out.println("<--");
17   }
18 }
```

Listing 2.5: Resulting code for inlining of around advice.

omitted in the listing for brevity. The difference is that, this time, a closure object is created and initialised. Then, the advice is called (on line 6 of the listing) and passed the closure.

The closure is implemented as an inner class of the base application class. It encapsulates the relevant state that the advice needs to be aware of (the target object of the method call, in this case). It has a `run()` method that is responsible for executing the join point shadow replaced by the around advice.

As can be seen from Lst. 2.6, the around advice is this time implemented in the aspect class. Instead of the `proceed()` statement, it invokes a dedicated method to which it also passes the closure. The proceed method invokes the `run()` method on the closure object, which ultimately leads to the execution of the original join point shadow.

The `aobj` array created in line 4 of Lst. 2.6 is populated with the state of the join point that is of relevance to the around advice. This state comprises of, e. g., `this` (if such an object exists) as well as the target and parameters of the join point wrapped in an around advice. In essence, a copy of the operand stack at the join point is made and passed to the around advice for further processing, and for being passed on to the original join point in case the advice proceeds.

F.3 `cflow` is handled using either counters or stacks. Counters are used when no state from the control flow "top" is required in the advice, or to match the dependent pointcuts. Stacks are used when state is required.

At `cflow` entries and exits, code is inserted into the base application that increments (or decrements, respectively) the counter associated with the control flow in question, or that updates the stack accordingly. At join point shadows depending on a control flow, a simple bit of conditional logic is inserted that checks whether the counter associated with the control flow is greater than zero, or if the stack contains state. If so, the associated advice is executed.

```
1  public class X {
2    public static void main(String args[]) {
3      X x = new X();
4      Object aobj[] = new Object[1];
5      aobj[0] = x;
6      A.aspectOf().ajc$around$A$1$8b20e847(new AjcClosure1(aobj));
7    }
8    ...
9    private class AjcClosure1 extends AroundClosure {
10     public Object run(Object aobj[]) {
11       Object aobj1[] = super.state;
12       X.y_aroundBody0((X)aobj1[0]);
13       return null;
14     }
15     ...
16   }
17 }
18
19 public class A {
20   ...
21   public void ajc$around$A$1$8b20e847(AroundClosure ajc_aroundClosure) {
22     System.out.println("-->");
23     ajc$around$A$1$8b20e847proceed(ajc_aroundClosure);
24     System.out.println("<--");
25   }
26   static void ajc$around$A$1$8b20e847proceed(A _this) throws Throwable {
27     Conversions.voidValue(_this.run(new Object[0]));
28   }
29   ...
30 }
```

Listing 2.6: Resulting code for around advice weaving using closures.

The control flow counters and stacks are basically instances of a subclass of `ThreadLocal`. The class `ThreadLocal` is a class from the Java standard library. It encapsulates an item of data and ensures that one specific instance of that item is created and maintained *per thread*. In Java 5, each instance of `Thread` maintains a `Map` that is used for storing all `ThreadLocal`s, where the `ThreadLocal` instance is used as the key and the thread-local item as the value.

G. Advice Instance Management

G.1 Advice instances are, for globally applying aspects, held in the aspect class itself. The `aspectOf()` method returns the only existing advice instance in this case. It is held in a static field of the aspect class.

Aspect instances are held either in the aspect instance or in the instance that matched a pointcut. In the case of perthis/pertypewithin the instances are held in the types that matched in a field with an internal name. In the case of persingleton the aspect instance is a singleton held in the aspect itself.

The instances are accessible via the aspectOf() method, variants of which take parameters depending on the aspect life cycle in use. For example, in the case of pertypewithin, the aspectOf() method takes a class indicating which class the caller would like to obtain

the aspect instance for.

G.2 AspectJ does not support scoping, i. e., the restriction of aspects to certain instances or threads. However, it allows for controlling the *instantiation granularity*. This means that advice instances are created *per* target object, for example.

In these cases, advice instances are held in the target classes, to which end the according fields for holding the advice instances are added to the target classes during weaving. The `aspectOf()` method of the aspect accepts an `Object` parameter in this case. It checks whether the passed object—the target instance—already carries an advice instance in the corresponding field. If so, that instance is returned; if not, it is created lazily, the field is initialised and the instance is returned. This process works analogously for `perthis` and other aspect instantiation granularity control mechanisms.

H. Dynamic Deployment Workflow

AspectJ does not support dynamic deployment.

I. Other Systems

Another prominent AOP language with strong compiler support and no dynamic weaving is AspectC++ [120, 18]. As much as AspectJ is an aspect-oriented extension to Java, AspectC++ extends the C++ programming language with AOP capabilities. The join point and advice models are very similar to those of AspectJ. The differences lie mostly in the syntax.

The `abc` Compiler

The AspectBench Compiler (`abc`) [124, 22, 1] is an alternative compiler for AspectJ sources[3]. Its primary goal is to provide a framework that allows for adding new features to the AspectJ language. Its open model of the language also supports the implementation of optimisations. Performance investigations conducted by the `abc` team have already led to the discovery of shortcomings in AspectJ's implementations of `cflow` and around advice [54] that have been mended subsequently.

Major contributions from the research on `abc` are optimised implementation approaches to both `cflow` and around advice that exceed the aforementioned mends, and the concept of *trace matches* [3], which are an extended version of stateful aspects (cf. Sec. 2.4).

In the context of this work, the static analysis techniques introduced for `cflow` are of particular interest because they address a problem specific to *dynamic* pointcuts with static means. Some intra-procedural optimisations—using counters instead of stacks for `cflow` matching when no context from the `cflow` is accessed, caching counters in local

[3] At the time of this writing, `abc` is available in version 1.1.0, which adheres mostly to AspectJ version 1.2.1. The Java 5 features described for AspectJ in this section are not supported in `abc`.

variables, and sharing `cflow` stacks for similar pointcuts—have been implemented in `abc` [22] that reduce the overhead of `cflow` management.

Still, the actual cost of monitoring `cflow` using counters and stacks is high. Using inter-procedural control-flow analysis, it can be reduced substantially. The analysis implemented in `abc` exploits a call graph of the entire application, which is why *all* classes of the application must be known at compile-time, or weave-time, respectively. For each pointcut expression containing a `cflow` designator, it yields three sets of join point shadows that are then further processed by the weaver.

Based on the example `cflow(pc1) && pc2`, the three sets computed are as follows (in the following, "residues" and "advice invocations" mean those pertaining to the sample pointcut only):

- The first set contains those shadows of `pc2` that *may* occur in a control flow initiated by a shadow of `pc1`. At the shadows contained in this set, advice invocations must be guarded by residues.

  At those shadows of `pc2` that are *not* contained in the first set, neither residues nor advice invocations need to be woven because they are guaranteed to never be executed inside a control flow pertaining to `pc1`.

- The second set contains those shadows of `pc2` that are guaranteed to occur *only* in a control flow initiated by a shadow of `pc1`. At these shadows, the advice invocation can be woven without being guarded by a residue.

  At those shadows of `pc2` that are *not* contained in the second set, residues are required.

- In the third set, those shadows of `pc1` (sic) are contained that *may* influence the evaluation of residues at shadows of `pc2`. At these shadows, residues for counter or stack maintenance must be woven.

While the optimisations gained through static analysis have led to significant speedups in applications using `cflow` [22], their impact on compile-time is large. The `abc` compiler, being slower than the AspectJ compiler anyway due to its optimising approach, again exhibits a performance degradation when the extensive inter-procedural analyses for `cflow` are applied.

A sample application[4] with 2 classes and 5 aspects (one of which using `cflow`) was used for a brief evaluation of compile-time performance. The AspectJ 1.2 compiler took 3.4 seconds, `abc` without static analysis took 19.8 seconds, and `abc` with aggressive optimisations took more than five minutes to compile the application.

`abc` imposes a closed-world assumption on application code it compiles. It performs a whole-program analysis: all aspects and Java classes that the application comprises of *must* be known at compile-time, and `abc` must be told the class containing the application's `main()` method. This is due to the the extensive inter-procedural analysis, which requires a complete call graph of the application. The possibility of dynamic class loading is not taken into account.

[4]The application was taken from from Chapter 9.4.3 of Laddad's book on AspectJ [108].

2.3. CaesarJ

A. Language Presentation

CaesarJ [111, 114, 115, 38] is an AOP language extending Java. Its focus is especially on modularity, reuse, flexibility and correctness. CaesarJ differs from other aspect-oriented languages in that it not only focuses on the physical separation of source code, to implement crosscutting concerns. It also ensures other important properties of modularity: abstraction, information hiding and minimisation of dependencies. Aspects are designed as components, which have clear abstraction and can be reused. In order to achieve these goals CaesarJ slightly extends run-time conception of object-oriented systems by grouping objects to collaborations, using virtual types and bindings. The main focus in the presentation below is on the aspect-oriented language features of CaesarJ contributing to the PA flavour of AOP.

The join point and advice models of CaesarJ are essentially those of AspectJ. CaesarJ also uses an AspectJ-like pointcut language. There is no special module construct for aspects in CaesarJ: pointcuts and advice are declared directly in CaesarJ classes (`cclass`es). Aspect objects are instances of such classes. Aspects have all properties of classes: instantiation, encapsulated state, inheritance and polymorphic usage.

Other than AspectJ, CaesarJ brings capabilities for dynamic deployment of aspects. The standard AspectJ behaviour for aspects—being active all the time—can be achieved by declaring a `cclass` to be `deployed`. Single aspect instances can be deployed and undeployed using the `deploy` and `undeploy` statements. An aspect instance's advice do not take effect until the instance is explicitly deployed. Deployment can be global, or it can be scoped to individual threads. Different instances of a `cclass` can be deployed on different scopes. The different approaches to deployment are provided in the form of so-called *deployment strategies*. It is possible to define custom deployment strategies.

Source code for the sample aspect is shown in Lst. 2.7. The class looks very much like an AspectJ aspect apart from its header. The `HelloAspect` class is marked as a `deployed cclass`, meaning that it is a class containing crosscutting definitions that should be *statically* deployed. In case the aspect defined by this class should be dynamically deployed, the `deployed` keyword is omitted.

```
1 public deployed cclass HelloAspect {
2   before(): call(void HelloWorld.hello()) {
3     System.out.println("I am an advice.");
4   }
5 }
```

Listing 2.7: Hello world aspect in CaesarJ.

B. Execution Model Architecture

B.1 CaesarJ is provided as a compiler and utilises the AspectJ weaver internally. CaesarJ programs run on a standard JVM. All necessary support classes are either generated

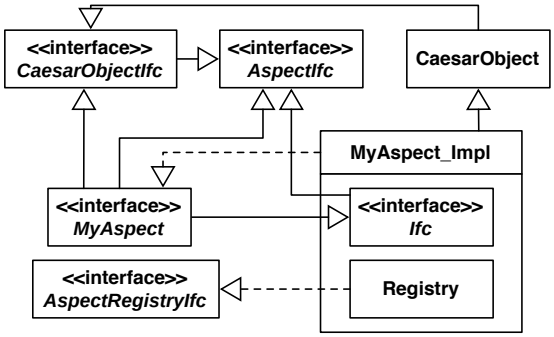

Figure 2.2.: Classes and interfaces involved in CaesarJ aspect representation.

by the compiler as a part of the program or available in the run-time libraries. The AspectJ run-time library contains the classes necessary to support the code woven by the AspectJ weaver. The CaesarJ run-time library mainly contains classes implementing different deployment strategies.

B.2 Application code is manipulated entirely at compile-time. At run-time, only conditional logic is executed, but no further weaving steps take place.

B.3 Like with AspectJ, the basic mechanism used to facilitate aspect behaviour is the insertion of advice invocations at compile-time. Other than with AspectJ, the CaesarJ weaver also inserts conditional logic that is responsible for checking the applicability for aspects at join point shadows. This is how dynamic deployment is achieved.

C. Programming Model Implementation

C.1 Any aspect class is compiled to a *set* of classes and interfaces by the CaesarJ compiler. For an overview of the classes and interfaces involved in the compilation of an aspect class named `MyAspect`, see Fig. 2.2.

An aspect class is, at run-time, basically represented by two Java classes: the aspect class and the registry class (`MyAspect_Impl` and `MyAspect_Impl.Registry` in the figure). Both are generated by the CaesarJ compiler, and they are not intended for direct use by the programmer. The aspect class implements the aspect's behaviour. The registry class manages the deployment of aspect instances.

Advice are represented as methods with internal names in the aspect class. Pointcuts have, as in AspectJ, no direct counterpart at run-time.

C.2 Aspects themselves are first-class entities in CaesarJ, advice and pointcuts are not. Dynamic assembly of aspects is not possible; all aspects have to be known at compile-time.

C.3 As with AspectJ, there is no protocol as such for aspects and advice to obey, since they are language entities. An important point is, however, that aspects can only be declared as `cclass`es.

D. Join Point Model Implementation

D.1 The application model used for join point exposure is the code of the application, in the form of its bytecode.

D.2 During weaving, the application bytecode is represented in BCEL [46, 26] entities.

E. Pointcut Model Implementation

Since the AspectJ weaver is utilised to perform pointcut evaluation, information about the join point shadow retrieval process can be found in Sec. 2.2.

F. Weaving Implementation

An overview of the weaving process can be found in Sec. 2.2. This presentation is restricted to the CaesarJ particularities.

F.1 Regardless of them being statically or dynamically deployed, *all* aspects are woven statically. Dynamic behaviour is simulated by registering and unregistering aspect objects at statically woven hooks.

F.2 Advice invocations work, in principle, like in AspectJ. In CaesarJ, however, they are *always* conditional since dynamic deployment has to be taken into account. For the sample aspect, the code woven in `HelloWorld.main()` looks like this:

```
1 public static void main(String[] args) {
2   HelloWorld h = new HelloWorld();
3   HelloAspect_Impl.Registry.aspectOf().ajc$before$1$4$0();
4   h.hello();
5 }
```

The semantics of the `aspectOf()` method in `HelloAspect`'s registry is the same as that of the very method in AspectJ. The implementation of the advice method differs, though. It can be seen in Lst. 2.8. The method first checks, in line 2, whether the aspect is deployed at all, i. e., whether *any* deployed instance of the aspect class exists. Next, it is checked whether the aspect is statically deployed (in that case, the `$singleAspect` field references the aspect instance). If so, the *actual* advice—implemented in the (synchronised) method `ajc$before$HelloAspect_Impl$4$0()`—is invoked. Otherwise—i.,e., if the aspect is *not* statically deployed—, the advice of all dynamically deployed instances are invoked (lines 7–13). Fig. 2.3 displays a sequence diagram of the invocation of the sample aspect's before advice attached to the call of `HelloWorld.hello()`.

```
1  public final void ajc$before$1$4$0() {
2    if($aspectContainer != null) {
3      if($singleAspect != null) {
4        $singleAspect.ajc$before$HelloAspect_Impl$4$0();
5        return;
6      }
7      Object inst[] = $aspectContainer.$getInstances();
8      if(inst != null) {
9        for(int i1 = 0; i1 < inst.length; i1++) {
10         Ifc aspObj = (Ifc)inst[i1];
11         aspObj.ajc$before$HelloAspect_Impl$4$0();
12       }
13     }
14   }
15 }
```

Listing 2.8: Implementation of advice invocation infrastructure in CaesarJ.

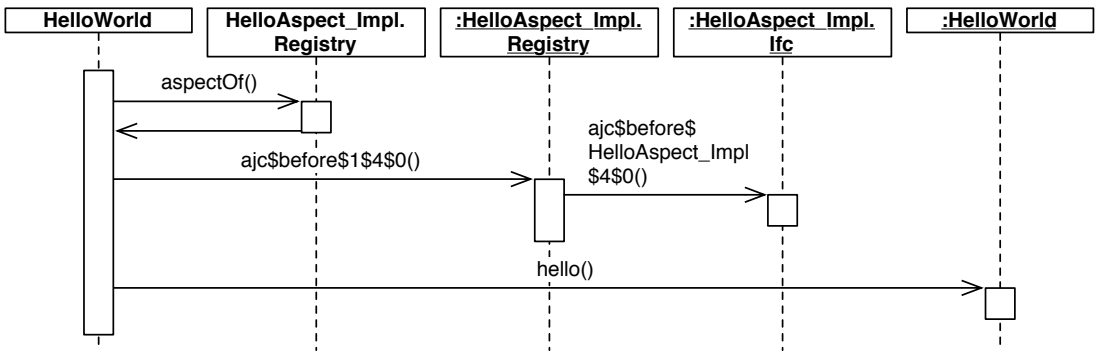

Figure 2.3.: Invocation of a before call advice in CaesarJ.

If the aspect is deployed dynamically only, the code of the advice method has exactly the same shape. The difference is that this time, the aspect instances to which the actual advice invocation is sent are retrieved from the aspect container.

F.3 For residues, the descriptions made for AspectJ in Sec. 2.2 apply. *All* residues, even those for instance-locally and thread-locally deployed aspects (cf. below), are implemented using standard Java facilities.

G. Advice Instance Management

G.1 Generally, CaesarJ aspect instances are maintained in the aspects' `Registry` classes. For aspect singleton instances—they exist, e. g., for statically deployed aspects—, the aspect implementation class maintains a static field that references the singleton. Nevertheless, the aspect instance is stored in the registry when the aspect class is initialised.

As long as only one instance of an aspect exists, it is directly referenced from a dedicated field in the registry. This is an optimisation that helps avoiding the management of complex data structures when only one aspect instance needs to be managed. As

soon as more than one instance of a given aspect exists, they are kept in so-called *aspect containers* referenced from the aspect's registry.

All aspect containers implement the `AspectContainerIfc` interface. Different container implementations exist for standard, thread-local and instance-local deployment. When queried, a container returns an array of all aspect instances it maintains.

G.2 CaesarJ supports thread-local deployment of aspects. There are two ways to programmatically deploy an aspect. With `deploy <aspect>;`, the aspect instance is deployed application-wide. When the `deploy(<aspect>) { ... }` form is used, the aspect is deployed in the given block, *and* it is scoped to the current thread. The aspect container responsible for thread-local deployment maintains a hash map with threads as keys and aspect instance arrays as values.

H. Dynamic Deployment Workflow

An aspect instance deployment, expressed in CaesarJ source code either via the `deploy` statement or as a parameter to a `deploy(){}` block, is compiled to an invocation of an according static deployment method in the class `DeploySupport`. That method forwards the deployment request to the appropriate *aspect deployer*, all of which implement the `AspectDeployerIfc` interface. In this interface, there are basically two methods defined; one for deployment and one for undeployment. Both accept one reference to an aspect registry and one to an aspect instance.

Default behaviour for aspect deployers is implemented in `BasicAspectDeployer`. Among other things, it takes care of maintaining the lazy creation of aspect containers, which becomes necessary when more than one instance of a particular aspect class exist (cf. above).

The deployer, responsible for normal (application-wide) deployment, simply adds the deployed aspect instance to the container. The thread-local deployer also controls which aspect registries actually contribute to thread-local aspects.

The effect of dynamic deployment basically is that the newly deployed aspect instance will be returned as a member of the list of aspect instances to send advice invocations to. This is done, e, g., in the code in Lst. 2.8.

Undeploying an aspect instance simply means to remove it from the containers.

I. Other Systems

The ObjectTeams [80, 122] follows a similar approach as CaesarJ. Its goal is also to provide a language abstraction for crosscutting collaborations that change dynamically. The capabilities of expressing crosscutting is however limited when compared to CaesarJ: the join point model is restricted to method invocations, and the pointcut language does not allow for quantifications.

The implementation of ObjectTeams is similar to that of CaesarJ. A compiler parses ObjectTeams and Java source code files and generates woven code. The woven code

realises dynamic weaving through conditional logic. No facilities of the underlying execution environment are used.

2.4. Arachne

A. Language Presentation

Arachne [51, 137, 66, 12] is an aspect-oriented extension to the C programming language. Its current implementation targets the x86 platform on Linux operating systems. Arachne's join point model is dynamic. The most basic join points supported are function calls, read and write operations on *global* variables, and read and write operations on variable aliases, i. e., references. Also, control flow can be matched.

A special pointcut language exists that not only formulates pointcuts, but also maps them to advice. Arachne has a very powerful pointcut named `seq` that matches *sequences* of join points. This feature stems from the EAOP language [52, 53, 57]. In EAOP, an event-based approach is is used to model join points: primitive join points that are reached signal an event, and a pointcut expression is an expression in an event algebra that represents a complex sequence of events. These concepts have been incorporated into Arachne's composite `seq` pointcut, which matches when the sequence of pointcuts it is composed of matches in exactly the given order.

There are two different control flow pointcuts. The semantics of the `controlflowstar` pointcut are equivalent to that of `cflow`. The `controlflow` pointcut is different in that it allows for the *precise* definition of call stacks. Using the latter, it is possible to define pointcuts that match only if a function is called from another that was in turn called from another, and so forth.

Advice in Arachne are ordinary C functions that are called when certain pointcuts match. They are directly mentioned in the aspect specifications. It is possible to attach multiple advice to a `seq` pointcut; more specifically, an advice can be attached to *every single pointcut* in the sequence. This leads to a sequence of advice executions during a matching sequence of pointcuts.

Arachne only supports around advice. They terminally *replace* the decorated join point shadows; there is no support for constructs like AspectJ's `proceed()`.

An aspect is a collection of pointcut-and-advice specifications, defined in a single file. It may define its own state and behaviour.

Arachne allows for dynamic weaving on running applications. Weaving is triggered from outside the running process, by invoking the appropriate application (the Unix applications `weave` and `deweave` are responsible for weaving).

An aspect is represented as a dynamic (shared) library. The weaver applications modify the binary code of the running base application and insert hooks into it that trigger the events needed by Arachne's join point model. These hooks branch execution, if a pointcut matches, to the advice defined as functions in the shared library.

Sample Aspect Since Arachne is based on C, the sample hello world application looks different for it (cf. Lst. 2.9). The sample aspect attaches an advice to the `hello()`

function—note that the original message printed by `hello()` will *not* be displayed since the call to the advice replaces the original code.

```
1 int main(int argc, char** argv) {
2     hello();
3     return 0;
4 }
5 void hello() {
6     printf("Hello, world!\n");
7 }
```

Listing 2.9: C hello world application for Arachne.

The aspect's source code is given in Lst. 2.10. The pointcut—`call(void hello())`—is specified in line 1, and the advice function is bound to it. Lines 3–5 contain the latter's definition; it is simply a C function without any syntactic specifics.

```
1 call(void hello()) then advice();
2
3 void advice() {
4     printf("I am an advice.\n");
5 }
```

Listing 2.10: Sample aspect in Arachne.

Example with Join Point Sequences The following example[5] is the implementation of an Arachne aspect that checks for buffer overflows.

```
1  seq(
2    call(void* malloc(size_t)) && args(allocatedSize) && return(buffer);
3    write(buffer) && size(writtenSize) && if(writtenSize > allocatedSize)
4      then reportOverflow(); *
5    call(void free(void*)) && args(buffer);
6  )
7
8  void reportOverflow() {
9    ...
10 }
```

This aspect invokes a function `reportOverflow()` if it detects a buffer overflow—i. e., a write operation to a buffer that attempts to write more bytes than the buffer is long—for a given buffer. The aspect consists of a sequence pointcut composed of with three pointcuts.

The first pointcut matches if the `malloc()` function is called to reserve memory for a buffer. The parameter to `malloc()` and its return value, i. e., the allocated size and a pointer to the buffer, are bound to the aspect-local variables `allocatedSize` and `buffer`.

[5]The example has been taken from [51].

Subsequent write operations to the same buffer are matched by the second pointcut. It binds the size of the data to be written to the buffer in the variable `writtenSize` and checks, using an `if` specialiser, whether the amount of data to be written is larger than the buffer size. If so, the `reportOverflow()` method is called.

This pointcut is marked with a `*` to denote that it may occur multiple times in the sequence, and that each of these occurrences has to be checked for an overflow. The sequence ends with a call to `free()` for the respective buffer.

B. Execution Model Architecture

B.1 Arachne's architecture is simple: there are two applications for weaving and deweaving aspects, and aspects are represented as shared libraries that are passed to the weaving applications. There is no infrastructure that is permanently running in the background of an application subject to decoration with Arachne aspects.

B.2 AOP mechanisms can be applied at any time while an application is running. The invocation of one of the weaving applications is equivalent to accessing AOP functionality.

B.3 The basic technique used by Arachne is to modify the application's binary code at those places that represent join point shadows, and to insert hooks there that signal events conforming to Arachne's join point model.

C. Programming Model Implementation

C.1 Each aspect is represented as a shared library (a Unix `.so` file). An application that is to be decorated with Arachne aspects has to run in an "Arachne context". This can be established by explicitly linking the Arachne kernel library with the application as it is started. If aspects are to be woven into an application that does not run in the Arachne context, the kernel library is dynamically loaded into the respective process space. The kernel library provides binary code rewriting and hook management logic that is needed during weaving and deweaving.

In the shared libraries, advice are represented as C functions. Since they are generated from compound instructions in the aspect definition, there is no protocol they have to adhere to with regard to their parameters. However, since all advice are around advice, they have to return a value of the same type as the replaced join point.

The Arachne aspect compiler, `acc`, compiles pointcuts to C functions that are, along with the advice functions, contained in the generated shared library. These special pointcut functions are executed when the aspect is woven. They modify the base application's native code by overwriting the loaded binary in memory, exploiting symbol table information. The applied modifications consist of inserting hooks at join point shadows.

For more complex cases where dynamic checks have to be executed, other C functions are created that serve the same purpose as AspectJ's *residues*.

C.2 Neither aspects, nor advice, nor pointcuts are available as first-class entities in a running application. Aspects cannot be assembled reflectively at run-time. They have to be compiled.

C.3 Aspects have to be implemented in the Arachne language. Apart from that, Arachne imposes no restrictions on the C functions used to constitute an aspect's behaviour.

D. Join Point Model Implementation

D.1 The application model used for join point exposure is the application's native machine code itself.

D.2 Arachne exploits the symbol tables of the applications it instruments to establish mappings between source code symbols and their addresses in running programs. Retrieving this information is expensive. Hence, Arachne supports caching the mapping information in so-called "meta-information DLLs" to speed up shadow retrieval operations in the future.

E. Pointcut Model Implementation

Join point shadow retrieval effectively takes place when an aspect is woven in. The base application's binary code is scanned for join point shadows of the aspect's pointcuts. Each join point shadow is an address in memory that is passed to the weaver.

For example, for function calls as well as global variable read and write pointcuts, the code is scanned for the appropriate assembler instructions for function calls and memory access. When such an instruction is found, it is checked to target the correct function or field. This is done by comparing the parameters to the assembler instruction with the memory address of the function or value in question.

F. Weaving Implementation

F.1 *Hooks* are woven into the base application. That is, the machine code instructions originally constituting a particular join point shadow are replaced with instructions that invoke a hook. The hook is responsible for performing residual checks and, in case they succeed, invoking the actual advice.

F.2 Advice are always invoked via the hooks inserted at join point shadows. Since the hooks are reached by a direct jump instruction, their execution is much cheaper than that of a function call.

Fig. 2.4 illustrates the control flow that is executed when a join point shadow decorated with an Arachne advice is reached[6]. Before the actual advice function, maybe prepended

[6]For Arachne, only the generic control flow is illustrated; the additional illustration for the sample advice is forgone.

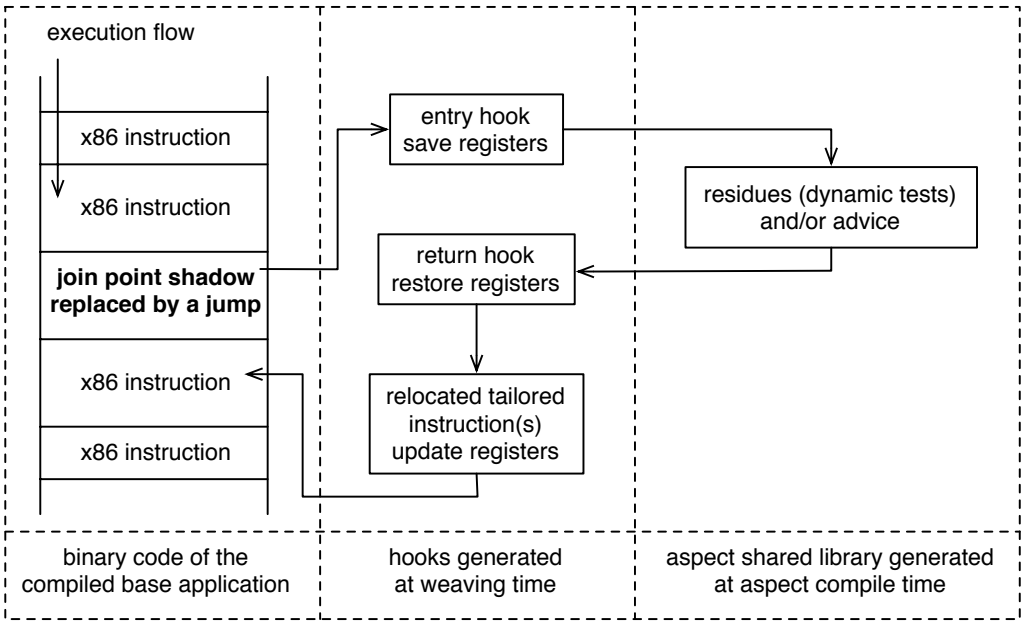

Figure 2.4.: The structure of woven code in Arachne (adapted from [51] by permission).

by residual code, is called, a hook is executed that saves the state found at the join point shadow. Once the advice function returns, the return hook restores the registers.

The additional code block executed after the returning hook has a special purpose. For example, consider a `read` join point shadow that is represented by a `cmp` assembly instruction. Such an instruction is typically followed by a conditional branch instruction that relies on the `cmp` setting some flags appropriately. The `cmp` instruction is replaced with a jump to the entry hook, which saves all processor registers. The return hook can however not take care of establishing the right state of the *flags*—this can only be achieved by actually performing a `cmp`. So, the "relocated tailored instructions" consist, in this example, of a `cmp` instruction that operates on a copy of the value returned from the advice to set the flags as needed by the following instruction.

F.3 To check for `controlflow` and `controlflowstar` pointcuts, the generated residual code looks up the required information in the stack frames. The entire stack is available to the residue since it operates at native code level.

The sophisticated `seq` pointcut is also matched by residues. For every `seq` pointcut, an automaton is created. At each join point shadow corresponding to one of the pointcuts contained in the sequence, a residue is woven that triggers a state transition in the automaton. To every state, an advice invocation may be attached. Whenever such a state is reached, the corresponding advice is called.

G. Advice Instance Management

There are no such things as advice instances in Arachne, since all advice invocations are ordinary C function calls.

H. Dynamic Deployment Workflow

Dynamic weaving approach that is triggered from outside the running application by invoking the `weave`/`deweave` applications. These are passed the PID of the application that is to be instrumented, and the shared library containing the aspect to be woven into the running process.

If the process is not already linked to the Arachne instrumentation kernel, this is done now. After that, the kernel invokes the various pointcut functions in the aspect library that, on their part, utilise the kernel's instrumentation capabilities to replace join point shadow instructions with hook calls. The hooks themselves are generated on the fly.

Deweaving simply replaces all hook calls with the original instructions.

I. Other Systems

The family of systems that Arachne belongs to is one of AOP implementations that weave directly on native code. Other systems in this family are TOSKANA [58] and KLAS [155]. Both TOSKANA and KLAS do however not target arbitrary running applications, but operating system kernels. Both systems allow for dynamically weaving aspects into a running kernel of the FreeBSD [65] (KLAS) or NetBSD [118] (TOSKANA) operating systems.

KLAS is aimed at supporting kernel profiling. Specifically, its intention is to support profiling approaches where the points in which a profiler is interested change at runtime. To be able to cooperate with KLAS, the kernel must be compiled with a modified version of the GNU C compiler [70] that is part of the KLAS suite. This compiler creates extended debug information symbol tables to allow for pointcut evaluation. KLAS aspects are defined in XML files that contain advice in C syntax.

TOSKANA was developed to support autonomic computing [104] properties at operating system kernel level. Unlike with KLAS, the kernel does not have to be compiled in debug mode; however, all optimisations such as inlining must be switched off for it to cooperate with TOSKANA. The TOSKANA toolkit brings a collection of C macros that allow for straightforward definitions of aspects. Aspects are, apart from the use of these macros, written in plain C and have a clear module concept: they are compiled as kernel modules that can be dynamically loaded or unloaded.

2.5. JAsCo

A. Language Presentation

JAsCo (Java Aspect Components) [144, 152, 64, 153, 89] is an AOP language targeted at component-based software engineering. Its focus is on providing reusable aspect

modules. The JAsCo language is a Java extension that introduces two new entities, *aspect beans* and *connectors*. Aspect beans allow for describing crosscutting behaviour independently of the base application using *hooks*. Connectors are used to deploy aspect beans in a concrete context and to specify combinations of aspect beans. Both aspect beans and connectors are described in dedicated files, for which JAsCo brings special compilers that must be used to translate JAsCo programs to Java bytecode.

Aspect beans are, in principle, regular JavaBeans. They are specified independently of concrete component types and APIs, making them highly reusable. Aspect beans contain one or more logically related *hooks* that describe the crosscutting behaviour. A hook includes a special kind of constructor that defines an abstract pointcut. A constructor receives several abstract method parameters that are bound to one or more concrete methods at the aspect's deployment time. Hooks also declare advice in dedicated methods that have the names of their application time (`before()`, `after()`, etc.).

Connectors are used to deploy abstract aspects in a concrete context. A connector allows to explicitly instantiate and initialise one or more hooks. The parameters passed to hook constructors make explicit at which particular join points a hook applies. JAsCo connectors also support wildcards to easily specify a range of join points for aspect application.

Normally, only one instance of an aspect is generated for a given instantiation expression. It is however also possible to automatically generate several aspect instances for a single expression. JAsCo supports creating particular hook instances per target object, class, method, and thread.

The JAsCo join point model is dynamic; it is based on that of AspectJ. However, it is restricted to join points pertaining to method invocations: field accesses *as such* are not supported. Apart from method call and execution join points, JAsCo supports join points specific to the JavaBeans programming model [91], namely the firing of JavaBean events and access to bean properties. The pointcuts in JAsCo support these join points, and moreover allow for designating restrictions (e. g., through `this`, `target`, or `cflow`).

Some restrictions that are not in the pointcut language but in the interfaces of aspect beans' hooks are possible. Aspects can be applied to specific instances, or a dedicated applicability condition can be provided using `isApplicable()` methods in hooks. The latter corresponds to AspectJ's `if` pointcut designator.

JAsCo also supports *stateful aspects*, which match sequences of join points. This concept was ported to JAsCo from the Arachne language (see Sec. 2.4 for a more detailed description). Stateful aspects are, in general, not covered in the presentation below.

The advice model in JAsCo is as powerful as that of AspectJ, supporting before, after and around advice. In some respects, it goes further. For example, around advice can be applied to method invocations that return instances or that throw exceptions of specific types.

JAsCo supports three types of weaving. Each of them is a dynamic weaving approach, but the ways in which dynamic weaving is implemented differ:

- the *preprocessor* approach inserts traps at all possible join points,

- with *run-time trap insertion*, traps are inserted and removed on-demand, depending on the installed aspects, and
- the *run-time weaver* physically inserts aspects in target classes, so that weaving, unweaving and reweaving take place entirely at run-time.

Because the run-time weaver is the most advanced of the three, the focus of the presentation below is solely on this approach.

The hello world aspect is implemented in two files, one providing the aspect, and one providing the connector. Their sources are given in Lsts. 2.11 and 2.12.

```
1  public class HelloAspect {
2    hook HelloHook {
3      HelloHook(method()) {
4        execution(method);
5      }
6      before() {
7        System.out.println("I am an advice.");
8      }
9    }
10 }
```

Listing 2.11: Hello world aspect in JAsCo.

The aspect in Lst. 2.11 defines a hook that can be applied to arbitrary parameterless methods, which is determined by the signature of the hook constructor in line 3. Inside the constructor, `method` is actually a parameter that can be used to define the pointcut the hook is attached to. In the example, the hook is applied to the execution of the passed method, since JAsCo does not support method call join points.

In the `before()` method, the functionality of the before advice pertaining to the hook is defined.

```
1  static connector HelloConnector {
2    HelloAspect.HelloHook h = new HelloAspect.HelloHook(void HelloWorld.hello());
3  }
```

Listing 2.12: Connector for the JAsCo hello world aspect.

The connector in Lst. 2.12 is static, meaning that it is to be deployed at load-time. Connectors that are not declared static can be used for dynamic weaving, which is however not supported by the run-time weaver. In the connector, an instance of the hook defined in `HelloAspect` is created. Its constructor is passed the method to which the hook applies.

B. Execution Model Architecture

B.1 The run-time weaver is a pure Java class library. It works entirely at run-time. The JPLIS (Java Programming Language Instrumentation Services) features of the Java

5 standard VM are utilised. Aspects are woven physically into target byte-code at run-time.

Access to AOP infrastructure is given either indirectly by putting new JAsCo connectors in a dedicated connector directory (the JAsCo run-time environment checks periodically for new connectors and deploys aspects accordingly) or by directly accessing the run-time infrastructure's API.

Bytecode manipulation is implemented using Javassist [40, 92].

B.2 The run-time weaver itself works *entirely* at run-time. However, since the JVM does not support schema changes, empty method stubs are inserted at load-time for all methods in loaded classes.

B.3 HotSwap [50] is used to perform dynamic redefinitions of methods subject to decoration with aspects.

C. Programming Model Implementation

C.1 JAsCo Aspect beans are compiled to Java classes. Aspect beans have, in their interface, a set of methods common to all aspects (they do not implement a common Java interface, however). These methods exist for the JAsCo run-time environment to be able to query an aspect bean for its hooks.

Hooks are compiled to classes implementing the `IHook` interface. `IHook` defines methods for aspect bean instance management, connector access, and advice execution. Methods pertaining to the latter are named `before()`, `after()`, and so forth. These methods implement the advice: they contain the advice code given in the hook source code. They accept parameters describing a join point (in the form of a `MethodJoinPoint`) and its context. A `MethodJoinPoint` encapsulates, among others, information on method parameters and results and provides methods for proceeding from around advice.

Connectors are compiled to classes implementing `Connector`. This interface defines methods that are mostly of interest to the JAsCo infrastructure. It also defines methods that allow for selectively activating and deactivating a connector.

Pointcuts, as mentioned above, consist of two parts: an abstract pointcut in terms of abstract method parameters, which is specified in the hook constructor at the language level, and concrete method signatures that are passed to hook constructors in connector definitions.

Internally, the abstract pointcut parts are instances of subclasses of the infrastructural class `PCutpointConstructorApplicationDesignator`. It has subclasses for pointcut designators like execution, call, target, cflow, etc., and for composite designators. The concrete method signatures are represented by instances of classes implementing `ISignatureMatcher`. This interface defines methods for matching join points and classes at run-time.

Both parts of the pointcut are first-class, and are merged when a connector is loaded into the system. They are referenced from the hook and connector classes that are generated by the JAsCo compilers.

C.2 JAsCo aspects remain first-class at run-time. They exist in the form of aspect beans, which are JavaBeans. Hooks and connectors also are first-class entities. Run-time assembly of aspects is not possible; their elements have to exist at compile-time.

C.3 Aspects can be arbitrary Java classes; they only have to define hooks. In the hooks, there exists some protocol for advice. Advice methods have certain names corresponding to their application at a join point (`before()`, ...).

D. Join Point Model Implementation

D.1 The exposure of join points in JAsCo is restricted to method invocations and executions. The application model is the application AST in the form of its bytecode. When stateful aspects are employed, an execution history model is also utilised to implement automata for matching aspect states.

D.2 Application bytecode is exposed through Javassist for both pointcut evaluation and weaving.

E. Pointcut Model Implementation

There are two events that may lead to pointcut evaluation: activation of a connector and class loading. The latter is important because an already activated aspect might apply to a newly loaded class.

Abstract pointcuts and method signatures are composed to join point matchers. The join point matchers visit a join point tree to select the appropriate join point shadows. The join point tree is lazily generated out of all join points that might be matched. Lazy generation avoids, for example, the generation of method nodes for a class when the class itself does not match. In addition join point trees are cached.

When a connector is activated, in principle *all* loaded classes are visited. Class loading only leads to the application of all activated connectors' matchers to that class.

F. Weaving Implementation

F.1 As mentioned above, the run-time weaver approach performs exactly *one* preparatory step at load-time. For each class that is loaded, all of its methods are copied, and an exact copy under a new name is generated and added to the class. This is necessary because the VM does not allow for adding methods to a class later. JAsCo needs to have method replacements to be able to generate woven code for method execution join points.

Weaving takes place when an aspect is deployed or undeployed. Apart from the preparation step mentioned above, this happens entirely at run-time. No weaving steps are performed at load-time unless a class being loaded is affected by some already deployed aspect. All of an aspect's join point shadows are immediately decorated with advice invocations.

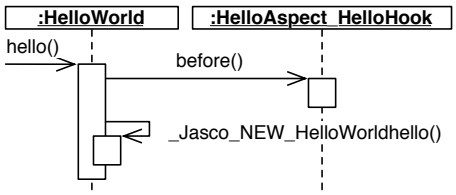

Figure 2.5.: Control flow for executing the sample advice in JAsCo.

The run-time weaver works on a per-class basis: it processes all advised join point shadows of a certain class at once. Because current VMs do not allow schema changes, a helper class is generated that contains external information, like the join point representations.

For method execution join points, a new method copy is created that contains the original byte code. This is also needed for around advice. Method call join points are redirected to advice, the remainder of the method byte code is not touched.

Hook instances are maintained in additional classes that are also transparently added at load-time. They also perform static initialisation of hook and join point information.

F.2 Advice are invoked directly. As an example, Lst. 2.13 shows how woven code looks for the sample aspect. The code has been slightly simplified. The body of `HelloWorld.hello()` has been replaced by woven code. In line 2, the hook is retrieved from the hook maintenance class, and the advice is immediately invoked in line 3. Line 4 contains the call to the method introduced at load-time that contains the original code of `hello()`. This control flow is also visualised in Fig. 2.5.

```
1 public void hello()() {
2   HelloAspect_HelloHook helloaspect_hellohook = _JAsCo_JuttaFieldClass1.hook1;
3   helloaspect_hellohook.before(Dummy.INSTANCE);
4   _Jasco_NEW_HelloWorldhello();
5 }
```

Listing 2.13: Woven code in JAsCo.

F.3 For matching `cflow`, a piece of conditional logic is inserted, and the execution of the advice is made dependent on it. The condition basically creates a stack trace by instantiating an `Exception` and attempts to find the method signature constituting the control flow therein.

G. Advice Instance Management

G.1 Aspects are instantiated explicitly—in the form of hooks—in JAsCo connectors, so it is possible that several instances of the same aspect live in the application. Per default, hook instances are kept as *static final* members of the connector class representation generated by the JAsCo connector compiler.

G.2 JAsCo allows for scoping aspects to specified instances. To that end, a connector must be declared non-static, and objects can be added to and removed from its scope via the `Connector.addInstance()` and `Connector.removeInstance()` methods.

It is important to note that the run-time weaver itself, at its current state of development, does *not* support scoping. The JAsCo run-time environment reverts to other execution strategies in case a non-static connector is met.

H. Dynamic Deployment Workflow

Deployment and undeployment always take at the connector level: connectors establish the relationship between the base application and aspect beans. Connectors are always added or removed as a whole. This means that, if aspect instances should be (un)deployed separately, they have to be defined in separate connectors.

Deployment consists of the following steps:

1. all applicable join point shadows for the aspects in the connector are identified,
2. all aspect instances instantiated in other connectors applicable to the selected join point shadows are identified (to detect possible conflicts),
3. weaving (or reweaving) is done at the target join points, and
4. the enclosing classes are redefined; JIT compilation is left over to the VM.

Undeployment basically consists of the same steps, with the difference that code once introduced is revoked again.

The run-time weaver does, at the time of this writing, *not* support dynamic deployment and undeployment. As mentioned above for instance-local deployment, JAsCo reverts to other execution strategies in these cases.

I. Other Systems

The author is not aware of any other systems that adopt JAsCo's implementation approach.

2.6. AspectWerkz

A. Language Presentation

AspectWerkz [21, 33] is a framework for aspect-oriented programming in Java. Its join point model is as strong as that of AspectJ, and it supports, like AspectJ, weaving at compile-time and load-time. It exceeds AspectJ, though, in that weaving at run-time is also supported. Applications built with AspectWerkz run on all standard-compliant JVMs.

In AspectWerkz, aspects can be defined in two ways:

1. using XML configuration files that define pointcuts and map advice to them, and

2. source code annotations, where aspects are implemented with annotated Java classes.

Apart from syntactic differences, these two styles are equivalent. For the remainder of this presentation, the Java 5 annotation style will be used.

The focus of this presentation is solely on AspectWerkz 2.0, and on the run-time weaving approach, which exploits the JVMTI agent interface of the standard JVM [101].

The sample aspect for the hello world application looks *exactly* like the one in annotation style presented in the section on AspectJ (see Lst. 2.2 in Sec. 2.2). However, the AspectWerkz load-time weaver needs to be instructed as to which aspects it has to weave into the base application classes. This is done using an XML definition like the one given in Lst. 2.14.

```
1 <aspectwerkz>
2     <system id="hello">
3         <aspect class="HelloAspect"/>
4     </system>
5 </aspectwerkz>
```

Listing 2.14: XML definition file for the hello world aspect in AspectWerkz.

B. Execution Model Architecture

B.1 AspectWerkz is entirely implemented in Java and exploits the Java 5 VM's HotSwap capabilities through the JVMTI; more specifically, the `-javaagent` option is used to attach a load-time instrumenting agent to the running VM. AspectWerkz's API is available as a collection of Java classes.

B.2 AOP mechanisms are applied at load-time and at run-time. At load-time, classes are prepared according to the definitions in an `aop.xml` file. Even if hot deployment is used, the respective join point shadows have to be prepared at load-time, through a so-called *deployment scope*.

Aspects can be deployed and undeployed at run-time. However, they cannot be "assembled" programmatically; they rather pre-exist as classes that are made cross-cutting by means of XML deployment descriptors (`aop.xml`), source code annotations, or hot deployment.

B.3 The basic technique is that of replacing the original method bodies with new ones that branch into the AspectWerkz infrastructure. No reflection is used in woven code.

C. Programming Model Implementation

C.1 In AspectWerkz, aspects are defined in *aspect systems*. In the scope of an aspect system, there can be aspects, mixins, and deployment scopes defined. Mixins are not regarded in this discussion.

There are definition classes for aspects, pointcuts, advice and other AOP concepts (`AspectDefinition`, `PointcutDefinition`, ...). They all are containers that hold the information relevant for the description of a given entity. An aspect, for example, is represented by its pointcuts, advice, deployment model, etc.

Pointcuts are represented as strings in AspectJ syntax that are contained in instances of `PointcutDefinition`. For pointcut evaluation purposes, they are converted to visitors that are used to determine whether a pointcut matches in a given code structure (class, or method).

Advice functionality is implemented as ordinary methods in Java classes, to which the corresponding `AdviceDefinition` instances refer. Advice invocations are invocations of such methods, eventually performed through the AspectWerkz infrastructure.

A deployment scope is represented by a pointcut without any advice attached to it. It serves for marking a certain number of join point shadows, namely those matched by the pointcut, for later decoration of advice to them.

C.2 Aspects are not available as first-class entities with a dedicated interface. However, AspectWerkz' internal data structures are not intended for direct use by a programmer. There is a collection of classes representing the meta-model AspectWerkz uses to represent aspects and other elements. Advice also are not available as first-class entities; however, *advice instances* (i. e., aspect instances) are. Pointcuts are not directly available as first-class entities.

C.3 Aspects do not have to adhere to a strict protocol: any class can define cross-cutting behaviour. The only "protocol" aspect classes have to obey is with respect to constructors: either a default constructor must be defined (or no constructor at all, in which case the standard default constructor is chosen), or a constructor must be given that takes an `AspectContext` parameter.

Methods constituting advice functionality can be arbitrary Java methods that have to adhere to some protocol with respect to the parameters they accept and the values they return.

- `before` and all kinds of `after` advice methods either accept no parameters, or they accept exactly one parameter, which is of the type `JoinPoint` or `StaticJoinPoint`. They return nothing (`void` type).

- `around` advice methods return `Objects` and accept a parameter that has either the `JoinPoint` or `StaticJoinPoint` type. In an `around` advice, proceeding is achieved by invoking the join point object's `proceed()` method.

Both `JoinPoint` and `StaticJoinPoint` are interfaces, where the former extends the latter. `JoinPoint` provides access to dynamic join point context information; its usage is therefore slower than that of `StaticJoinPoint`.

D. Join Point Model Implementation

D.1 The application model used to expose join points is the application's bytecode, and its meta-model in the form of information about classes and methods.

D.2 General information about classes and methods is retrieved at load-time from the class file directly as it is being loaded. For this purpose, and for the analysis and modification of method bytecode instructions, AspectWerkz uses the ASM bytecode toolkit [17]. ASM represents Java classes using a low-level abstraction of a `byte` array.

E. Pointcut Model Implementation

Join point shadows are retrieved using visitors that are applied to classes, methods, etc. These visitors are responsible for both *matching* join point shadows as they visit a structure and *weaving*, i.e., matching and weaving are done in one step.

Join point shadow retrieval takes place in two phases during weaving. In the first phase, called "early matching", a quick pass is made over an entire class to find out whether specific kinds of join point shadows occur at all in the class. This is done to avoid the creation and application of transforming visitors in vain, and it actually constitutes an optimisation by avoiding "eager" filtering of entire classes. In the second phase, various visitors are created based on the results from the "early matching" phase, and then they are applied to the class.

The visitors used during join point shadow retrieval and weaving conform to the interfaces defined by the ASM toolkit.

F. Weaving Implementation

In the weaving model regarded in this survey, application classes are instrumented at load-time no matter whether an aspect is fully woven at load-time or whether join point shadows are merely prepared for later weaving through a deployment scope. Once an application class is loaded, it is never modified again.

Weaving is done by means of the Java 5 instrumentation API. An agent (installed in the VM by means of the `-javaagent` option) installs a preprocessor that is responsible for initiating class transformations as classes are loaded. This is done based on the aspect definitions found in the system. Aspect definitions are parsed whenever a class loader is initialised.

F.1 During weaving, classes are transformed according to aspect and deployment scope definitions. The effect of weaving is that all instructions falling under a join point shadow are *replaced* with calls to dynamically generated classes of the `JoinPoint` type. For each shadow/method combination, such a `JoinPoint` subclass is created on the fly.

F.2 The code that replaces the original shadow instructions is a method call to the `JoinPoint`'s static `invoke()` method that either directly executes the original shadow

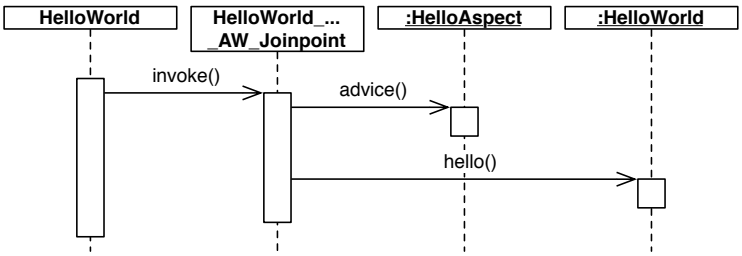

Figure 2.6.: Control flow in AspectWerkz for the sample hello world aspect.

(if the join point shadow was merely prepared) or that invokes advice functionality and the original shadow (if an aspect was woven in). A sequence diagram for the sample aspect is shown in Fig. 2.6.

The original shadows are, in case of `get/set` and `execution` shadows, moved to internally-named new methods that are introduced during weaving. "Execution of the original shadow" in that case means that the `JoinPoint` invokes one such method.

So, in a nutshell, advice are invoked entirely without the use of reflective mechanisms. However, the indirections through introduced `JoinPoint` classes, as described above, apply. There are at most two such indirections, namely if advice are applied that belong to hot-deployed aspects (see below). Normally, i.e., in case of a statically-woven aspect, there is only one indirection.

F.3 When an aspect containing a pointcut following the idiom `pc1 && cflow(pc2)` is woven, all join point shadows of `pc1` are decorated in the usual way. The important difference is that the `JoinPoint.invoke()` implementations now check whether the control flow has been entered, and that they only invoke advice functionality if that is the case.

To maintain control flows, another class is created dynamically. It is a subclass of `AbstractCflowSystemAspect` that is unique for this particular control flow. This cflow helper maintains a stack to monitor execution. The `JoinPoints` for the `pc1` shadows query this class for the aforementioned checks.

The join point shadows of `pc2` are also advised, which means that they too are replaced with a `JoinPoint.invoke()` call. This particular implementation of `invoke()` is special in that it wraps the execution of the original shadow in a call to the cflow helper class's `enter()` and `exit()` methods, thereby notifying the helper of the state of the execution.

G. Advice Instance Management

G.1 Aspect instances are stored in `JoinPoints`, and the `invoke()` implementations directly access them. This holds for aspects that are deployed "per JVM" or "per class".

During weaving, a method `aw$getAspect()` is introduced to every instrumented class, along with a member called `aw$InstanceLevelAspects`. These are used to store and retrieve aspect instances as used when aspects are deployed "per instance". In that case, the `JoinPoint.invoke()` implementation does not directly reference its aspect instance

field, but instead retrieves the instance through calling `aw$getAspect` on the respective object.

G.2 It is possible to mark certain kinds of join points as "advisable". Classes to whom this applies are transparently instrumented to implement `Advisable`. At run-time, it is then possible to attach advice to *single instances* of the instrumented classes, and to pass arbitrary pointcut expressions in that process. Advice classes, in this case, have to adhere to some protocol. They must implement a specific interceptor interface, which effectively means that they have to provide an `invoke()` method.

Shadows marked as *advisable* are replaced with `JoinPoint` invocations like all prepared shadows. The `JoinPoint.invoke()` implementation however is different in this case: it *iterates* over all attached advice and invokes them, eventually also invoking the original shadow. During instrumentation, methods to add and remove advice are added to a class containing shadows marked as *advisable*. These methods simply forward to the corresponding methods in the `JoinPoint` which stores the advice.

Scoping aspects to threads is not supported directly in AspectWerkz.

H. Dynamic Deployment Workflow

Hot deployment only works for join points that have been prepared at load-time using deployment scopes. If an aspect is hot-deployed, the respective application class is not modified. The `JoinPoint.invoke()` method that normally just invokes the original shadow functionality is now treated as the locus of the join point shadow.

This means that, for the aspect being woven, new `JoinPoint` subclasses are generated on the fly, as for weaving at load-time. Calls to their `invoke()` methods are inserted in the `JoinPoint.invoke()` methods that were generated at load-time, when the join point shadows were merely prepared. This insertion takes effect through redefinition of the `JoinPoint` classes through the JVMTI.

Undeployment works the other way around, meaning that advice invocations belonging to the aspect being undeployed are removed from the shadow-preparing `JoinPoint` class, which is then redefined.

I. Other Systems

There are numerous AOP implementations that are conceptually related to AspectWerkz. AspectJ 5 [43] is a joint venture of the developers of AspectJ and AspectWerkz. It still differs from AspectWerkz in that it does not support dynamic weaving, but AspectJ 5 has support for load-time weaving.

Other prominent systems that are conceptually close to AspectWerkz are JBoss AOP [93] and JAC [127, 88]. They both support dynamic weaving, and they both instrument classes at load-time according to aspect definitions. They have not been taken into account in this work because of the large similarities of their approaches to that of AspectWerkz.

2.7. PROSE

A. Language Presentation

PROSE (PROgrammable extenSions of sErvices) [128, 129, 119, 130] is dedicated to dynamic aspect-oriented programming on Java programs. It was one of the first AOP implementations that offered dynamic weaving capabilities.

Aspects in PROSE are defined programmatically, there is no language extension with a dedicated compiler. PROSE's dynamic join point model is about as powerful as that of AspectJ; it supports field access, method entry/exit (*not* call), and exception throw/-catch join points. Also, `cflow` is supported. The advice model is restricted to before and after advice.

The hello world sample aspect is expressed in two classes, one representing the aspect and one the "cut", or pointcut, which also encapsulates advice functionality. The aspect is shown in Lst. 2.15. It serves as a container that knows which crosscuts it manages. During deployment, the PROSE system queries the aspect for its crosscuts via the `crosscuts()` method.

```
1 public class HelloAspect extends Aspect {
2     protected Crosscut[] crosscuts() {
3         return new Crosscut[] { new HelloCut() };
4     }
5 }
```

Listing 2.15: Hello world sample aspect in PROSE.

Actual crosscutting and crosscutting behaviour are described in the `HelloCut` class, whose source code is shown in Lst. 2.16. It defines two methods: `pointCutter()` returns a description of the pointcut to whose join point shadows this cut's functionality is to be attached. The functionality is defined in the `METHOD_ARGS` method.

```
1 public class HelloCut extends MethodCut {
2     protected PointCutter pointCutter() {
3         return Executions.before().AND(Within.method("hello"));
4     }
5     public void METHOD_ARGS(HelloWorld target) {
6         System.out.println("I am an advice.");
7     }
8 }
```

Listing 2.16: The "cut" of the PROSE sample aspect.

Finally, the application itself, making use of PROSE aspects, looks different. In Lst. 2.17, the contents of its `main()` method are given. The PROSE subsystem has to be explicitly started up for an aspect to be inserted. The aspect is instantiated and deployed using the `insert()` method of the aspect manager. From that moment on, it is active, and executions of the `hello()` method are accordingly decorated.

```
1 public static void main(String[] args) {
2     ProseSystem.startup();
3     HelloAspect aspect = new HelloAspect();
4     ProseSystem.getAspectManager().insert(aspect);
5     HelloWorld h = new HelloWorld();
6     h.hello();
7     ProseSystem.teardown();
8 }
```

Listing 2.17: Main method of the hello world application for use with PROSE.

`HelloCut` extends the `MethodCut` class, denoting that it will be attached to method execution join points. The `PointCutter` returned from the first of its methods builds a more precise description: the advice is attached *before* the execution, and the name of the method being executed must match the string "hello".

Further restrictions are made by `METHOD_ARGS`' formal parameter: it is a `HelloWorld`, denoting that this advice should *only* be executed when the "hello" method being entered is defined in the class `HelloWorld`.

B. Execution Model Architecture

B.1 PROSE has a two-layered architecture. The *AOP engine layer* takes care of aspect management, advice execution and other high-level tasks that are independent of a specific implementation technique of low-level mechanisms. The latter are provided by the *execution monitor layer*, which is responsible for join point generation. It notifies the AOP engine of join point occurrences, to which the AOP engine then can react by, e. g., executing advice functionality. The AOP layer also explicitly asks the execution monitor layer to install/uninstall specific join points as needed by the aspects managed by it.

The AOP layer is intended to be used by a programmer. It is delivered as a pure Java API. While it is quasi-standardised, there exist, in the present version 1.3.0 of PROSE, several different implementations of the execution monitor that directly reflect on the basic mechanisms used:

- The JVM's standard debugger API can be employed to reify join points as debugger breakpoints ("debugger-based weaving"). This model is implemented using a small VM plugin written in C that is used to establish a connection from the debugger to the PROSE infrastructure.
- Since JDK 1.4, the JVM supports HotSwap [50] to modify the bytecodes of loaded classes while the application is running (the VM has to be run in debug mode for the feature to be enabled). This way, methods containing join point shadows can be selectively recompiled. The "advice weaving" approach of PROSE utilises this technique.
- In Java 5, the JVMTI (Java Virtual Machine Tools Interface) [101] was introduced. It facilitates HotSwap without the need for running the VM in debug

mode. HotSwap via the JVMTI also serves as a basis for implementing the advice weaving approach.

- IBM's Jikes Research Virtual Machine (RVM) [6, 4, 7, 8, 97][7] was used to implement another approach. The JIT compiler of the Jikes RVM was modified to insert calls to the PROSE infrastructure (which then checked for advice applicability) at all potential join points ("hook-based weaving").
- The advice weaving approach is also integrated in the Jikes RVM. To that end, the VM was extended to support selective method recompilation.

In the following discussion, the focus is on debugger-based and advice weaving only.

B.2 PROSE allows for (un)deployment of aspects entirely at run-time. They can however not be freely assembled while the application is running; the corresponding classes must be present as compiled classes when the application is started. All AOP mechanisms are applied at run-time; no specific steps are performed during compilation or class loading.

B.3 The mechanisms employed to facilitate AOP functionality are basically the same for all of the aforementioned implementation approaches of the execution monitor. In all cases, the application (or the VM executing it) is made to notify the AOP layer of the occurrence of a join point. In the case of *debugger-based weaving*, a breakpoint is registered that, when reaches, branches execution to the AOP layer. When *advice weaving* is used, a call to the AOP layer is directly woven into the application code.

C. Programming Model Implementation

C.1 *Aspects* in PROSE are instances of subclasses of the abstract `Aspect` class. They reference a number of `Crosscut` instances which in turn define the aspect functionality along with the pointcuts at which it is applied.

`Crosscut` classes couple pointcuts and advice. The `Crosscut` class has various specialised subclasses for field accesses, method invocations, exception throws and catches, and method redefinitions. User-defined aspect functionality is implemented in heirs of these classes.

Pointcuts are represented by instances of classes from the `PointCutter` hierarchy. `PointCutter` is the root of a hierarchy of classes that are primarily *filters*. They are subdivided into two categories, namely *insertion-time* and *run-time* filters. As their names suggest, the former are applied when an aspect is deployed (inserted), while the latter are applied when join points are encountered while the application is running. Insertion-time filters are based on static properties of the application (i. e., on its code) can be used to restrict the execution of advice to specific classes, methods, etc., based on decisions such as "field is defined in class `X`", "execution is in method `m()`", and so forth.

[7] An overview of this virtual machine will be given in Sec. 3.3.

In contrast, run-time filters apply to dynamic properties of the running application, such as the current `this`, the target object of a method invocation, etc.

Advice functionality is strongly coupled to pointcut definitions. They actually contribute to the task that is normally taken over by pointcuts: they determine, refining the definitions made by the `PointCutter`s in a crosscut, at which exact join points they apply. This is achieved by several conventions that advice have to adhere to. For example, an advice method applying at a method entry or exit must be named `METHOD_ARGS`. Its parameter list is important: if it is, for example, `(MyClass x, int y)`, the advice will apply *only* to methods invoked on an instance of `MyClass` that accept one `int` parameter.

C.2 First-class support for aspects is given: the instances of `Aspect` subclasses can be passed around. By designing the `crosscuts()` method in a dynamic way, aspects can even be assembled dynamically. Pointcuts and advice are not as strongly supported. Objects representing them can be obtained in the form of `Crosscut` instances, but dynamic assembly is not straightforwardly possible.

C.3 The PROSE programming model brings with it a strong protocol. Aspects must, as mentioned above, extend the `Aspect` class and implement the `crosscuts()` method. This method returns the set of `Crosscut`s that constitute an aspect. In `Crosscut` subclasses, the *pointcut* of a given crosscut is returned from the crosscut's `pointCutter()` method.

The protocol for advice is very strong. Methods constituting advice functionality return `void`, they must have a specific name (depending on the join point type they respond to), and their signature determines at which specific join points they apply (cf. above).

D. Join Point Model Implementation

D.1 The application model used for join point exposure is different depending on whether the debugger-based or advice weaving model is used. For *debugger-based weaving*, join points are exposed to the AOP infrastructure as events that are raised whenever a certain method is invoked or a field is accessed. The events originate in the underlying execution layer. When *advice weaving* is used, the application's AST, i. e., its bytecode representation, is used to expose join points.

D.2 The two different weaving approaches also use different techniques to access the application model. In *debugger-based weaving*, it is the JVM's debugger API that is used to register breakpoints. To know at which bytecode indices breakpoints have to be registered, a BCEL [26, 46] representation of classes is created and parsed for join point shadows. *Advice weaving*, being integrated with the VM, works on the VM's internal representation of bytecodes, but it utilises BCEL to have a more convenient abstraction than `byte` arrays.

E. Pointcut Model Implementation

Join point shadow *retrieval* is naturally only performed for filters that are applied at insertion time. During request generation (cf. below), each crosscut constrains the set of classes for which it generates requests according to the applicable filters. Basically, each generated `JoinPointRequest` (subclasses of which exist for field get/set, method entry/exit, and so forth) constitutes one retrieved join point shadow.

The application model that is used at insertion time is the model of the application that the standard Java reflection classes provide. Since at this stage not single instructions, but high-level concepts (such as "field set", "method entry", etc.) are used to represent join point shadows, the Java reflection model is sufficient even though it is not very fine-grained.

F. Weaving Implementation

F.1 When *debugger-based weaving* is used, no application code is ever modified. Instead, debugger breakpoints are registered at the appropriate join points that are represented by `JoinPointRequest`s (cf. below).

The case is more complicated for *advice weaving*. In this case, the bytecode instructions of the methods containing join point shadows are instrumented to perform a call into the PROSE infrastructure at the respective join points. For each kind of join point, there exists a specific callback method in PROSE. The code inserted into the application method bytecodes is minimal; it solely consists of a few instructions to set up context information and invoke the callback.

Both weaving approaches have in common that weaving (in this case, the activation of join points) takes place during the insertion of aspects into the running system.

F.2 As written above, advice are eventually invoked through the PROSE infrastructure. The part of the infrastructure that provides the callbacks is part of the *execution monitor* and therefore dependent on the weaving model. In both cases, some context information is passed to the respective callback. The callback classes implement `JVMAspectInterface`, which serves as the interface of the execution monitor to the AOP engine.

For *debugger-based weaving*, a `JoinPoint` instance is prepared in the VM plugin (i. e., in native code) and passed to the appropriate callback method implemented in `AspectInterfaceImpl`. The callback then forwards execution to the `JoinPointManager`, which is part of the AOP engine layer.

For *advice weaving*, the callback is also a method in the respective implementation of `JVMAspectInterface`. (The class implementing the interface is `AdviceJVMAI` in this case.) It is the responsibility of the callback to set up a `JoinPoint` instance that can be passed to the AOP engine layer. Once that is done, execution is forwarded to the `JoinPointManager` as above.

The remainder of the advice execution is in the hands of the AOP engine layer. It iterates over all `Crosscut` instances applying at the given join point shadow and notifies

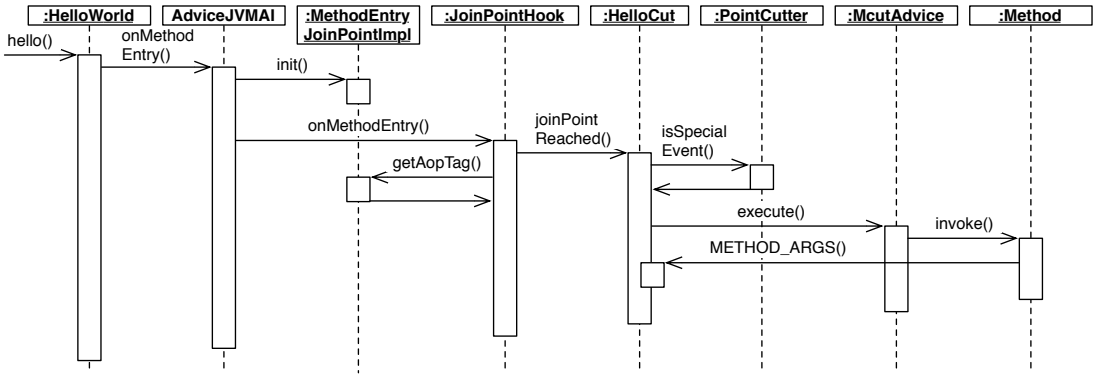

Figure 2.7.: Control flow of advice execution for the PROSE sample aspect.

them of the join point occurrence through their `joinPointReached()` method.

In `joinPointReached()`, the various crosscut classes apply the run-time filters and proceed with advice execution only if they match. The last step performed is the retrieval of the appropriate advice method and its invocation. To achieve this, a closure is created and the advice is invoked reflectively.

An overview of the control flow eventually leading to advice invocation for the sample aspect is shown in Fig. 2.7. It is simplified in that the entire process of matching the method against the pointcut is only represented by the invocation of `isSpecialEvent()` on a `PointCutter` instance.

F.3 Dynamic pointcuts, including `cflow`, are implemented as filters in PROSE. During the preparation of an advice execution, filters are applied; and each of the filters applying at a given join point shadow is asked to perform its matching operation.

For `cflow`, the filter subsequently retrieves the join point's *enclosing* join points and checks whether the control flow matches. All filters that have something to do with name matching, e.g., filters that dynamically check whether `this` is an instance of a given class, use *regular expressions* to perform matching.

G. Advice Instance Management

G.1 Advice instances in PROSE are effectively the instances of the various `Crosscut` classes that are defined by an aspect. During deployment (cf. below), they are attached to the join points they apply to as "listeners".

G.2 PROSE supports scoping aspects to single objects, but not to single threads. For scoping aspects to objects, a dynamic filter (e. g., `Target.inCollection()`) can be used that is passed a `Collection`. At run-time, the filter checks whether the collection contains the respective object.

H. Dynamic Deployment Workflow

Aspects are deployed by invoking `AspectManager.insert()`, passing it an `Aspect` subclass instance. Subsequently, the PROSE infrastructure retrieves all `Crosscut`s from the aspect instance and generates `CrosscutRequests` from them. A `CrosscutRequest` is merely a `Vector` to which less generic and more fine-grained requests are added. They are represented by instances of `JoinPointRequest`.

The `JoinPointRequest`s are created based on constraints that are defined by insertion-time filters (cf. above). Once all such requests have been created, the `JoinPointManager` is asked to register the crosscut instances as listeners to the join points. This leads to the join points being actually "activated" as such, which process is once more dependent on the particular execution monitor implementation. The *debugger-based weaving* implementation simply registers breakpoints whenever a join point is activated.

When *advice weaving* is used, the methods in which join point shadows lie are identified. The `JoinPointRequests` denote which methods are affected. These methods are cloned, and invocations of the appropriate AOP layer callbacks are woven into their bytecodes. After that, the method clones are installed in the VM. Subsequent invocations of these methods lead to the clones being executed. Advice weaving follows a "transactional" approach in that it cumulates all modified methods and reinstalls the woven code at the very end of the aspect insertion.

It is possible that a particular join point is used by more than one crosscut. In order to avoid duplicate activations or premature deactivations, mappings are maintained, ensuring that a join point, once activated, is never overridden and not deactivated before the last aspect using it is withdrawn from the system.

I. Other Systems

At the time of this writing, BEA [27] is working on a version of its JVM implementation JRockit [100] that offers support for dynamic AOP. This implementation is not publicly available.

Conceptually, its AOP model [85, 86] is close to that of PROSE: so-called *subscriptions* can be attached to certain events in program execution. Subscriptions are endowed with *filters* that match class, method and field names. The JRockit AOP API allows for fully dynamic weaving.

While the notion of attaching filters and subscriptions to run-time events is something both PROSE and JRockit AOP have in common, the actual programming model in JRockit AOP strongly differs from that of PROSE. JRockit AOP's filters are based on the Java language's reflective model. Aspects in JRockit AOP are assembled programmatically from subscriptions and filters; advice are ordinary Java methods. The strong programming protocol observed in PROSE does not exist in JRockit AOP.

2.8. Spring AOP

Spring [98, 140] is a J2EE [87, 116] framework that avoids the use of EJB containers. It facilitates using simple Java classes conforming basically only to the JavaBeans [91] protocol as J2EE entities. In other J2EE implementations, services like persistence and transactions are provided through EJB containers that are complicated to configure. Spring follows an entirely different approach in that it introduces services transparently through aspect-oriented programming. The AOP framework used for this is Spring AOP [141].

A. Language Presentation

The Spring AOP framework is intended for use in conjunction with Spring itself, so the design philosophy of Spring has influenced the design of Spring AOP very much. In consequence, interfaces play an important role in Spring AOP as well as in overall Spring. AOP entities in Spring AOP are implemented in classes conforming to the AOP Alliance [9] interfaces.

Aspects are defined by assembling instances of specific classes representing pointcuts and advice. This frequently takes place in XML files, as usually with Spring. Aspects are actually JavaBeans, which makes them easily serialisable and manageable through the bean management facilities upon which Spring is built.

The join point model of Spring AOP is simple; it only knows method invocations. Field accesses are deliberately not part of the join point model because they break the encapsulation principles of object-orientation. Moreover, being able to advise field accesses would break the "interface only" approach of Spring AOP that is closely connected with the JavaBeans philosophy. The support for advice is not restricted: all kinds of advice—before, after, around—are supported.

Spring AOP is implemented using facilities that the Java programming language provides. There are no language extensions at all. The dynamic proxy capabilities that were introduced with Java 1.3 are used to implement method call interception and advice execution.

The hello world application looks different from the version presented in Sec. 2.1.2 when it is used with the Spring framework because of the latter's strong use of interfaces. The application itself is shown in Lst. 2.18. Most notably, the application class now implements the `IHelloWorld` interface.

In the `main()` method, a so-called "application context" is created based on the XML specification shown in Lst. 2.19. From the context, an instance of `HelloWorld` is retrieved that is decorated with the aspect as configured in the XML file.

In the XML file, first a bean with the name `helloBean` is defined—it is exactly the bean that is retrieved in the `main()` method of the application. The bean is defined to be generated by the `ProxyFactoryBean` class. Lines 4 and 5 are very important: they define the interface that the proxy should implement, and the bean that provides the *actual* implementation. In line 10, this implementation bean is defined to be an instance of the `HelloWorld` class.

```
1  public interface IHelloWorld {
2      void hello();
3  }
4
5  public class HelloWorld implements IHelloWorld {
6      public static void main(String[] args) {
7          ApplicationContext ctx =
8              new FileSystemXmlApplicationContext("hello.xml");
9          IHelloWorld h = (IHelloWorld) ctx.getBean("helloBean");
10         h.hello();
11     }
12     public void hello() {
13         System.out.println("Hello, world!");
14     }
15 }
```

Listing 2.18: Hello world application and its interface for Spring AOP.

```
1  <beans>
2      <bean id="helloBean"
3        class="org.springframework.aop.framework.ProxyFactoryBean">
4          <property name="proxyInterfaces"><value>IHelloWorld</value></property>
5          <property name="target"><ref local="helloWorldImpl"/></property>
6          <property name="interceptorNames"><list>
7              <value>advisorBean</value>
8          </list></property>
9      </bean>
10     <bean id="helloWorldImpl" class="HelloWorld"/>
11     <bean id="advisorBean"
12       class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
13         <property name="advice"><ref local="adviceBean"/></property>
14         <property name="mappedNames"><list>
15             <value>hello</value>
16         </list></property>
17     </bean>
18     <bean id="adviceBean" class="HelloAdvice"/>
19 </beans>
```

Listing 2.19: XML definition for the sample aspect in Spring AOP.

The advisor responsible for the proxy is set in lines 6–8; it is the `advisorBean` defined in lines 11–17. This bean maps the execution of the method `hello()` to the `adviceBean`, which is, in line 18, defined to be an instance of the `HelloAdvice` class.

B. Execution Model Architecture

B.1 As mentioned above, Spring AOP does not comprise any language extension. Its execution model is essentially that of the Java programming language. No modifications to compilers or the run-time environment are made in any way. There is no interference with class loaders as well.

Spring AOP is provided as a framework. The framework contains a set of interfaces to which aspects must conform, and a set of JavaBean classes that represent containers for aspects and pointcuts. Moreover, some support classes are provided that take care

of, e. g., pointcut evaluation.

B.2 Spring AOP can operate entirely at run-time. It is possible to assemble all elements of an aspect while an application is running. Of course, methods that are to be used as advice have to be existing when they are composed to yield an aspect, but composability is unlimited.

Even though the capabilities of Spring AOP allow for an entirely dynamic use, aspects are—conforming to the rules applied when Spring is adopted—normally specified in configuration files and explicitly activated from inside the application.

Weaving in Spring AOP is fully dynamic.

B.3 Spring AOP uses *dynamic proxies* [55] to implement advice invocations. Dynamic proxies were introduced as part of the Java language standard libraries when Java 1.3 was released. A dynamic proxy is an object that intercepts all method invocations sent to the instance it wraps. The interfaces to which a dynamic proxy conforms are chosen at its creation time.

C. Programming Model Implementation

C.1 A Spring AOP *aspect* actually only defines a set of mappings of pointcuts to advice, represented as *interceptors*, or *advisors*. An aspect is represented as a dynamic proxy. For its creation, usually the `ProxyFactoryBean` is populated with several properties:

- `proxyInterfaces` denotes all interfaces the proxy intercepts,
- `target` references the object (another bean) for which the proxy stands, i. e., that is decorated by the aspect, and
- `interceptorNames` enumerates the various advisors defined by the aspect (and, hence, mappings from pointcuts to advice).

The aspects, or dynamic proxies, respectively, are instances of dynamically created subclasses of the `target` object's class.

An aspect encapsulates, as mentioned above, several advisors, each of which is a bean conforming to the *Advisor* interface. Several classes implement this interface; the `RegexpMethodPointcutAdvisor`, for example, maps a pointcut containing wildcards to an advice. This bean class has properties named `advice` and `patterns`. Such an advisor can specify name patterns for the methods its advice applies to.

There exist advisor classes for the special purpose of *dynamically* matching method invocations. These advisors allow for implementing `cflow` and other dynamic pointcuts.

Advisors have a hidden property containing their pointcut. *Pointcuts* are represented by instances of classes implementing the *Pointcut* interface. Instances of these classes are created implicitly when advisors are created.

Advice are represented by beans whose classes implement the `Advice` interface. Below this interface, which is empty end exists for tagging purposes only, there is a hierarchy

of interfaces for different purposes, namely for representing before, after returning and after throwing advice, and for representing *interceptors*, or around advice.

C.2 Basically every entity in a Spring application is a JavaBean. Aspects, pointcuts and advice are no exception from this rule. Aspects have however no clear *aspect* interface: they are proxy instances and not of any type that actually suggests them being aspects.

Dynamic assembly of aspects at run-time is well possible, but the classes that are used for this have to be preexisting. The mostly used approach in Spring AOP is however to give the configuration in an XML file and load it at run-time.

C.3 Due to Spring's strong adherence to interfaces that also reflects on Spring AOP, there is a considerable amount of protocol that parts of aspects must adhere to.

Concrete advice classes must implement one of the corresponding interfaces and provide an implementation of the advice method that is defined in each of them. There are strict rules for its interface:

- Before advice classes implement `MethodBeforeAdvice`, whose `before()` method accepts

  1. a `Method` object representing the called method,
  2. an `Object` array containing the (possibly auto-boxed) parameters to the call, and
  3. an `Object` representing the call target.

  The `before()` method's return type is `void`.

- After returning advice implement `AfterReturningAdvice`; their `afterReturning()` method accepts the same parameters as `before()` above, and an additional `Object` parameter referencing the return value. Its return type is `void`.

- After throwing advice implement `AfterThrowingAdvice`. Their `afterThrowing()` method accepts the parameters known from before advice, and an additional `Throwable` subclass instance representing the thrown exception. Its return type is `void`.

- Around advice implement `MethodInterceptor`. Its `invoke()` method is different from all of the above in that it accepts only a single parameter of the type `MethodInvocation`. Its return type is `Object`. The `MethodInvocation` interface provides functionality for retrieving context information and proceeding with the original method call.

It is noteworthy that none of the Spring AOP advice have full access to method parameters and return values. They can only manipulate the respective objects via their interfaces but cannot entirely replace them.

D. Join Point Model Implementation

D.1 The join point model met in Spring AOP is very simple in that only method invocations are regarded as join points. This reduces the application model that can be queried for join point shadows to the interfaces of the available classes.

D.2 Exposure of the join point model is entirely done via Java's reflective capabilities.

E. Pointcut Model Implementation

Join point shadow retrieval is not a very complicated process in Spring AOP, since the only possible join points are method invocations. Any aspect is clearly configured in a way that allows for straightforwardly resolving all join point shadows, i. e., all methods whose invocations have to be intercepted by a proxy.

In general, `MethodMatcher`s are responsible for checking whether a method matches a pointcut. `Pointcut` instances reference such matchers; they are queried when proxies are created, or at run-time if dynamic matching has to be performed (cf. below).

Pointcuts that can be statically evaluated are those that directly enumerate a set of methods (or method patterns) which they are intended to match. Such pointcuts are normally instances of a subclass of `AbstractRegexpMethodPointcut`[8], and they are queried when proxies for statically defined methods are created.

Dynamically evaluated pointcuts are represented by instances of subclasses of the classes `DynamicMethodMatcherPointcut` or `ControlFlowPointcut`. The latter is a specialised pointcut class for `cflow`; the former can be used to create custom pointcuts that dynamically match method invocations based on arbitrary decisions.

These pointcuts are not evaluated at proxy creation time. Rather, the proxies are set up in a way that makes them invoke the `matches()` method of the pointcut implementation in question. This returns a `boolean` value denoting a match, in which case the proxy will proceed with the advice invocation.

F. Weaving Implementation

F.1 Neither classes nor methods are transformed *at all* during weaving since Spring AOP uses proxies.

F.2 Essentially *all* advice in Spring AOP are invoked as around advice due to the nature of advice implementation using proxies. This holds even for before and after advice. When a method call is executed, it is the proxy to whom the invocation is sent.

The proxy maintains an interceptor chain for all advice that are attached to a method invocation. For each element of the chain, the appropriate before, after, and around advice are executed in the given order. Eventually, the original method is invoked reflectively.

[8] The concrete class depends on the variant of regular expressions that is chosen to match method names.

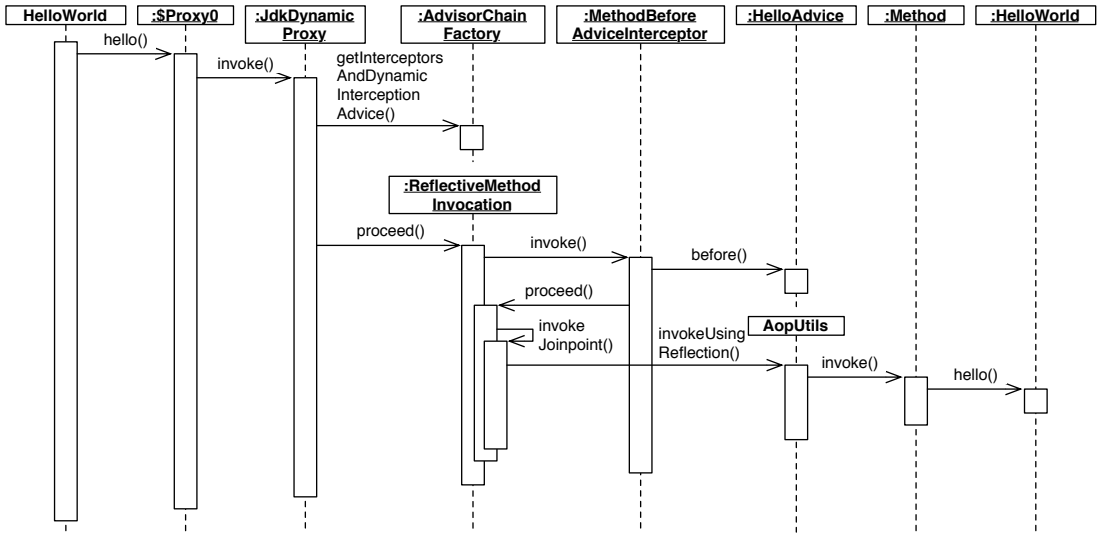

Figure 2.8.: Control flow for executing a before advice in Spring AOP.

The complexity of the control flow associated with a before advice is illustrated for the sample aspect in the sequence diagram in Fig. 2.8. The fact that all advice—even before advice—are practically executed as around advice is evident from the control flow originating in the `ReflectiveMethodInvocation` instance in the center of the figure.

F.3 Residues are implemented in the `matches()` methods of the pointcut classes capable of dynamic matching. As mentioned above, `DynamicMethodMatcherPointcut` can be subclassed by the user to specify user-specific dynamic pointcuts. The class responsible for matching `cflow`s is `ControlFlowPointcut`. It follows a straightforward approach in that it creates a `Throwable` instance and iterates over the stack trace contained therein to match the control flow.

It should be noted that the kind of support Spring AOP has for `cflow` is semantically not the same as usually met. In Spring AOP, a `cflow` pointcut matches when the control flow originates in some *class*. It is not possible to give methods whose execution constitutes a control flow.

Dynamic matchers are evaluated during the iteration over the interceptor chain, immediately prior to the eventual invocation of the corresponding advice.

G. Advice Instance Management

G.1 In general, it is controlled at object creation time whether it is decorated with an aspect or not: it depends on whether a proxy is assigned to an object or not. Aspects are normally *not* applied to all instances of a class, but only to those objects that have explicitly been decorated with the corresponding proxy.

However, there exists a possibility to control scoping in a more fine-grained way. Proxies retrieve the target objects they decorate from a `TargetSource` instance. There

are various types of `TargetSource`s that allow for, e. g., changing the object that is decorated by a proxy on the fly (`HotSwappableTargetSource`), or creating a target object per thread. Another target source, `PrototypeTargetSource`, creates a new target object upon each request sent to the proxy.

In a nutshell, there is no way to implicitly achieve the decoration of *all* instances of a class with an aspect in Spring AOP. Those instances that are explicitly decorated with a proxy are subject to aspect behaviour when method invocations are sent to them. Others are not.

G.2 As mentioned above, Spring AOP aspects are inherently scoped to single instances. Scoping them to threads is not directly supported by Spring AOP. It has to be emulated using a `DynamicMethodMatcherPointcut`.

H. Dynamic Deployment Workflow

Dynamic deployment of a Spring AOP aspect actually consists in creating an appropriate proxy for subsequent invocations to be sent to. From the configuration, provided via the Spring configuration XML file for instance, the set of interfaces the aspect should intercept is retrieved, and the proxy is created accordingly.

Next, the proxy's interception methods are set up according to the information that is retrieved from the advisors defined in the configuration, and the interceptor chains for the various join point shadows are set up. The target source is configured and, if needed, the target object is created.

Finally, the newly created proxy class is passed to the class loader and an instance is created. This is returned for further use by the programmer.

As for the decoration of single *objects* with aspects, an object is, as mentioned above, either decorated at its creation time by being instantiated through a Spring AOP proxy factory, or by being made a target object explicitly. Deployment simply consists of making the object available from the corresponding `TargetSource`.

There is no actual undeployment step in Spring AOP. An aspect can however be made ineffective by extracting the target objects from it and henceforth using them directly instead of calling them via the proxy.

I. Other Systems

Proxy-based AOP is comparatively easy to implement, since standard interfaces like the dynamic proxy API can be exploited. Hence, quite a number of such AOP implementations have been introduced over the last few years. Spring AOP is the most prominent and actively developed of them. Three other examples are Nanning [117], DynAOP [56] and Jeet [94]. However, none of these appears to be under active development at the time of this writing.

Nanning is based on the standard proxy API. Aspects are represented as mixin modules consisting of an interface, a target object and a number of interceptors for the methods declared in the interface. Aspects are defined using source-code annotations or

XML definitions. It is also possible to decorate objects with aspectual behaviour programmatically, in which case the appropriate interceptors are attached to every single method of the object's interface.

Nanning supports about the same range of features as Spring AOP. However, Nanning requires interfaces and implementations to be separated, if they shall be subject to decoration with aspects. Spring AOP does not have this kind of restriction.

The DynAOP implementation is based on the same principles. However, it supports, like Spring AOP, the decoration of functionality that is declared in *classes* rather than interfaces. DynAOP is special in that its approach to aspect definitions is done through BeanShell [28] scripts. By using such scripts, aspect definitions can be set up in Java syntax using the same API that the AOP infrastructure exhibits. The API is rich and allows for expressing comparatively complicated pointcuts.

Jeet has full support for Java 5 annotations to define aspects, apart from XML and programmatic assembly. Like Spring AOP, Jeet's implementation conforms to the AOP Alliance interfaces. In its features, Jeet does not differ much from Spring AOP. The major difference is that it does not have a JavaBeans-centered philosophy and that it is not, like Spring AOP, a natural part of a larger framework for multiple purposes.

2.9. Reflex

A. Language Presentation

Reflex [148, 146, 149, 147, 135, 145, 133, 134] is a kernel for multi-language AOP on the Java platform. It facilitates the implementation of different aspect-oriented languages so that it is possible to compose aspects written in these different languages. Since Reflex is built on concepts whose terminology is somewhat alien to AOP, the introduction to this section is more extensive than those to the other related work descriptions.

Originally, Reflex has been a toolkit for *partial behavioural reflection* [149] in Java. Its creators have realised that such reflection can be a valid means to implement AOP functionality, and have further on developed it as a "versatile AOP kernel" [148, 147, 135] that can be used to implement various AOP languages by providing required low-level mechanisms.

The core concepts of Reflex are provided around an implementation of *partial reflection* for Java. In reflective programming, entities constituting a program are *reified*—i. e., made available as first-class entities to the program itself. Reflection is *partial* if only those elements that are of actual interest are reified, instead of all possible elements.

Reification means that, for a given point in an application, a meta-object representing that point is created. When execution reaches the point, a *hook* is executed that follows a *link* to the meta-level, i. e., a branch, to execute additional behaviour defined there. Hooks can be grouped in *hooksets* that share some common properties. The execution of behaviour in the meta-object may depend on *activation conditions* attached to the link. These concepts and their relations are shown in Fig. 2.9.

In Reflex, so-called *cuts* are used to describe places in code where hooks are attached. There exist *structural* and *behavioural* cuts, corresponding to descriptions of static and

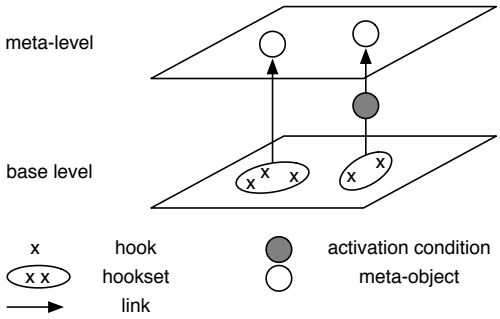

Figure 2.9.: Concepts of Reflex (adapted from [149]).

dynamic crosscutting in AOP. Consequently, there also are *structural* and *behavioural links* (s-links and b-links, for short), responsible for binding structural and behavioural cuts, respectively, to actions implemented in meta-objects.

Structural cuts typically describe sets of classes, and structural actions describe the modification of classes, e. g., adding members to them. Behavioural cuts describe sets of points in the execution of a program, and behavioural actions describe augmented behaviour. The discussion of Reflex in this work focuses on behavioural cuts and links, since they establish dynamic crosscutting.

A behavioural cut is implemented as a *hookset*. Primitive hooksets are associated to specific Java operations with class and operation selectors, responsible for selecting the classes and, within these classes, the operation occurrences of interest. Primitive hooksets can be combined into composite hooksets.

It is possible to attach so-called *activation conditions* to links. A branch of execution flow to the meta-level is only performed if an activation condition attached to a link evaluates to `true`. *Meta-object protocols* (MOPs) attached to the links describe what functionality is to be invoked on meta-objects in what specific way when a link is activated, i. e., when reification occurs at a given point in the program.

To implement AOP languages, AOP concepts are mapped to the aforementioned Reflex concepts in the following way. *Pointcuts* are mapped to cuts, i. e., hooksets. *Advice* are represented by actions implemented in meta-objects. Bindings between pointcuts and advice are established through links. *Residues* of all kinds are implemented using activation conditions. *Aspects* basically are collections of links.

Using Reflex, aspects are defined in two parts:

1. A *link provider* has to be implemented that defines the cuts in the application, and the links from hooksets to meta-objects where aspect behaviour is implemented. It is possible to have more than one link providers applied to an application. They normally come in the form of Java classes defining cuts and links programmatically.

2. The aspect functionality has to be provided in the form of normal Java classes.

Details on these points will be provided below.

A Reflex application is started by invoking the Reflex harness implemented in the class `reflex.Run`. It is passed the configuration classes (i. e., the link providers) and the actual application to run.

The sample aspect for the hello world application basically consists of a trivial class containing the advice method (see Lst. 2.20), and of a configuration class that sets up the links accordingly.

```
1 public class HelloAdvice {
2     public void advice() {
3       System.out.println("I am an advice.");
4     }
5 }
```

Listing 2.20: A class containing the advice method for the Reflex sample aspect.

The configuration is shown in Lst. 2.21. The `initReflex()` method is responsible for establishing the desired behavioural links from hooksets to the meta-level. The hookset used for the sample aspect is the set of hooks attached to all calls to `HelloWorld.hello()`. The `PrimitiveHookset` representing this set is created in lines 3–7 of the listing. The hooks are attached to message sends (`MsgSend`) that may originate in all classes (`AllCS`, "all-classes selector") and are directed to `HelloWorld.hello` (determined by the instance of `CallSelector` that is created).

```
1  public class HelloConfig extends ReflexConfig {
2      public void initReflex() {
3          PrimitiveHookset h = new PrimitiveHookset(
4                  MsgSend.class,
5                  AllCS.getInstance(),
6                  new CallSelector("HelloWorld", "hello")
7              );
8          BLink l = addBLink(
9                  h,
10                 new MODefinition.MOClass("HelloAdvice")
11             );
12         l.setScope(Scope.CLASS);
13         l.setControl(Control.BEFORE);
14         l.setActivation(Activation.ENABLED_START_ON);
15         l.setMOCall(new CallDescriptor(
16                 "HelloAdvice",
17                 "advice",
18                 CallDescriptor.NO_PARAMS
19             ));
20     }
21 }
```

Listing 2.21: Reflex configuration for the sample aspect.

In lines 8–11, a behavioural link is created and attached to the hookset, which is passed as the first parameter to the `addBLink()` method. The second parameter denotes the class whose instances are intended to be meta-objects connected to the link at the meta-level. After its creation, the link is further set up: its application scope is set to `CLASS`

so that it can be attached to hooks originating in static methods, it is declared to be effective *before* the hook is executed, and it is activated.

Finally, in lines 15–19, the `advice()` method defined in the meta-object class is established as the action to be taken when the link is followed at run-time.

B. Execution Model Architecture

B.1 Reflex is implemented as a Java class library. It runs on any standard JVM. AOP mechanisms are facilitated through a customised class loader that processes aspect definitions and triggers the appropriate weaving operations. Access to the AOP functionality of Reflex is provided through its API.

An application subject to decoration by Reflex's AOP mechanisms must be started through a special Reflex harness. The harness is passed configuration information in the form of classes defining aspect configurations (link providers, cf. above).

B.2 Base application code is prepared for the later attachment of advice to join point shadows at load-time. During this step, calls representing links are woven into base application code at join point shadows matched by hooksets given in the link provider. Weaving is achieved through Javassist [40, 92].

In Reflex terminology, the step undertaken at load-time performs partial reflection with *spatial selection*: it selects the *elements* to be reified. Later, at run-time, *temporal selection* allows for selectively activating and deactivating hooks. An invocation to the meta-level is performed only if a join point shadow, i. e., the hook attached to it, is activated.

B.3 The basic technique for the implementation of AOP mechanisms are *hooks* that branch to the meta-level and spawn further functionality, e. g., residues and advice. In Reflex terminology, these are represented by activation conditions and methods of meta-object classes.

C. Programming Model Implementation

C.1 Aspects are, as described above, defined in two parts, a link provider and the classes that actually constitute aspect behaviour. The link provider is internally represented as a class, references to which are passed to the Reflex harness at application startup time. Aspect behaviour is implemented in Java classes.

In general, an aspect will internally be represented as a *linkset*, i. e., sets of instances of classes implementing the `BLink` interface. The `LinkGroup` class represents such sets. Pointcuts are represented by hooksets that in turn are represented by instances of subclasses of the `Hookset` class. Subclasses exist for primitive and composite hooksets.

Links have several attributes pertaining to their application. Their `scope` attribute indicates whether they apply per instance, per class, or globally. Their `control` attribute indicates in what way, relative to the execution point in question, control is passed to the meta-level (before, after, or around).

The *MOP descriptor* of a link defines precisely how the linked meta-object should be invoked. This invocation can be guarded by an *activation condition* or a *hookset restriction*. The difference between a restriction and an activation condition is that a restriction is embedded within base code, and therefore cannot be manipulated at run-time, whereas an activation condition is an object referenced at the base level, which can be accessed dynamically. Due to this, activation conditions are used to implement complex residues (cf. below).

An advice is typically implemented as a method in the interface of a meta-object class. Internally, it is represented by a `CallDescriptor` that encapsulates a method call and parameter passing information. `CallDescriptor`s are referenced from links to which they are bound.

C.2 Aspects and advice are not available as first-class entities. Meta-objects, i. e., advice instances, are accessible, though. Hooksets are first-class entities at load-time only.

Links are true first-class entities. Any link can be obtained in the form of an instance of a class implementing the `RTLink` (run-time link) interface. `RTLink` provides read/write access to the meta-objects associated with the link and allows for its dynamic activation and deactivation.

C.3 A link provider class inherits from `ReflexConfig`, which is itself a subclass of `LinkGroup`. The crucial method for setting up links that has to be implemented by a link provider is `initReflex()`. In this method, the links are defined by assembling them from hooksets and action descriptions.

A class constituting aspect behaviour and state does not have to adhere to any protocol. The same holds for the signatures of methods used as advice. There is an exception, however, for around advice. Methods constituting around advice accept one parameter of the type `IExecutionPointClosure` and return an `Object`. The closure parameter passed to them is the object to which such advice can send the proceed message, and which they can access to modify parameters of the original join point.

User-defined activation conditions implement the `Active` interface, which requires one method to be provided: `public boolean evaluate(Object)`. The parameter is the object in which the hook that triggered the condition evaluation was met. The `evaluate()` method is required to return `true` if the link to which the activation condition is attached should lead to a reification.

D. Join Point Model Implementation

D.1 The application model used to expose join points is the application's meta-model. The meta-model is, however, only partially reified. Those parts that are reified are basically in bytecode.

D.2 The bytecode is exposed to the Reflex infrastructure at load-time through the Javassist API. Javassist allows for load-time instrumentation of bytecode.

E. Pointcut Model Implementation

The evaluation of pointcuts in Reflex consists of finding those places in application code that match hooksets. The hooksets that apply to a program are known at startup-time. The links that contain them are registered with the Reflex class loader, which in turn intercepts class loading. Whenever a new class is about to be loaded, the Reflex loader checks which of the links it knows has any interest in the class.

All links that match the class are gathered, and once all links have been checked for applicability, a `HookFactory` is created. Subsequently, an iteration over all particular operations—such as message receptions, field reads/writes, etc.—that are contained in one of the matching links is started. `HookInstallers` are created and passed to the `HookFactory` that then modifies the class accordingly.

Join point shadow retrieval in Reflex boils down to matching hooks to application bytecodes, which are immediately instrumented. After a class has been loaded, *all* statically resolvable join point shadows have been woven. Since Reflex does not explicitly distinguish between statically and dynamically resolvable shadows—it only knows about hooks, links to which *may* be subject to activation conditions—, basically all shadows are statically resolvable.

F. Weaving Implementation

F.1 Weaving is done through bytecode transformation at load time. During the weaving phase, each class matched by a hookset is modified in the following way:

- The class is made to implement the `RObject` interface, which provides several methods for meta-object access. Implementations for these methods are added to the class. They are for internal purposes.
- For each link from the class to the meta-level, members representing the link, activation conditions, and the meta-object are added to the class. Also, for each link, several management methods are added that are responsible for, e. g., activation checks and lazy meta-object initialisation.
- In each place in the code of the class where a hook applies, hook code (cf. below) replaces the original code.
- The bodies of methods to which a message reception hook applies are replaced with new bodies just consisting of a hook (cf. below), and the original bodies are moved to new methods with internal names.

Hook code consists of an activation check, a meta-object lookup and an advice invocation. The latter two are executed only if the activation check evaluates to *true*. In Lst. 2.22, the (simplified) code for the before hook from the sample aspect presented above can be seen.

The calls to the `IN()` and `OUT()` methods exist for marking purposes only; both methods are empty. In line 2 of Lst. 2.22, the activation check is performed. The called

```
1 CodeWrapper.Mark.IN();
2 if(_CheckActive_HelloConfig_B_1()) {
3     _CheckInitMO_HelloConfig_B_1();
4     ((HelloAdvice)_MO_HelloConfig_B_1).advice();
5 }
6 h.hello();
7 CodeWrapper.Mark.OUT();
```

Listing 2.22: Simplified hook code generated for the sample aspect.

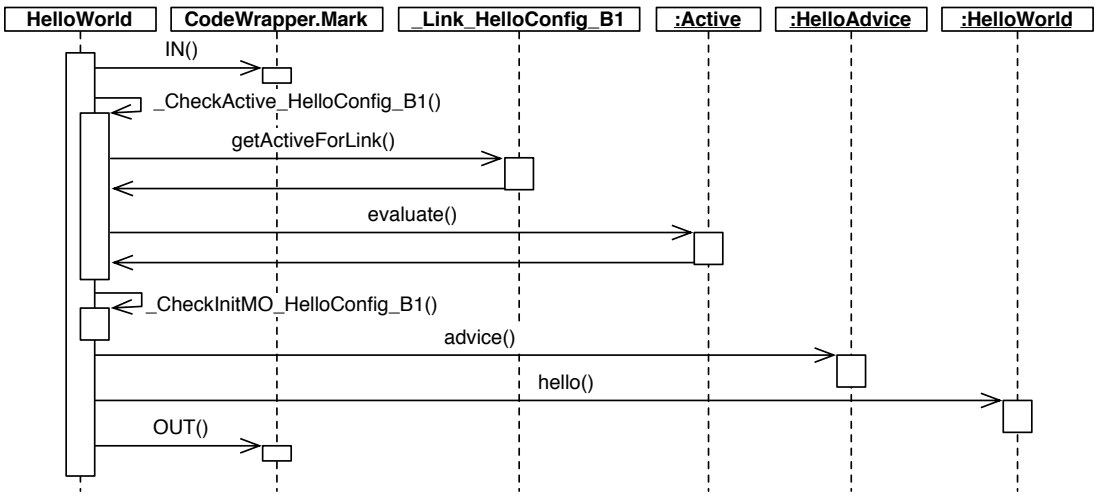

Figure 2.10.: Control flow for executing the sample aspect in Reflex.

method checks whether the link is activated at all, and also evaluates the activation condition attached to the link, if any. In line 3, the initialiser for the meta-object is invoked. It simply checks whether the object has already been created and does so if this is not the case. Line 4 contains the advice invocation. In line 6, the decorated call is performed.

F.2 Advice invocations are executed directly on the advice instances. For around advice, the invocation is preceded by the setup of the execution point closure that is passed to the advice.

The control flow executed when an activated link is on hand is shown in Figs. 2.10. The sequence diagram reflects the sample aspect.

F.3 Residues are generally implemented through activation conditions. Their capabilities to perform complex evaluations on execution state are however limited, since the only parameter passed to an activation condition is the object in which the corresponding hook was just met. Information that is required to perform, e. g., `cflow` matching cannot be retrieved from this source object.

In case a construct like—in AspectJ syntax—`cflow(pc1) && pc2` is used, there are

two pointcuts corresponding to *two hooksets*: one hookset corresponds to the pointcut constituting the control flow, and the other one to the dependent pointcut. In Reflex, two links are created and connected in the following way to implement `cflow`:

1. A link for the control flow is created using Reflex's `CFlowFactory`. Its hookset is the one corresponding to `pc1`. The hooks are connected to meta-objects that maintain control flow counters.

2. Another link for the dependent join points is created in the normal way. However, a special activation condition is attached to it, namely a `CFlowActivation`. This class has two subclasses, `IsInside` and `IsOutside` that match when a control flow is currently on the stack, or when it is not.

Through the activation condition, the link corresponding to `pc2` is only reified if one of its hooks occurs inside (or outside) the control flow that the link is attached to. The activation conditions are simple: they query the connected control flow's counters.

G. Advice Instance Management

G.1 Advice instances are meta-objects. They are kept in fields in the base-level classes that are added during the weaving step performed at load-time. Such a field exists per link from a particular class to the meta-level. Depending on the scope of a link (cf. above), there are various ways to store advice instances:

- If a link is *globally* scoped, each class includes a shared reference, initialised via a global link repository.

- A link with *class* scope leads to each class having its own reference in the form of a static member variable.

- In case of *object* scope, each object has its own meta-object reference.

G.2 As seen above, Reflex does not support scoping in the full sense of this work, i. e., the applicability of aspects to single objects or threads, but aspect instantiation control. Scoping has to be emulated through activation conditions.

A very limited support for scoping is available, though. Through the selective activation and deactivation of links for single objects (cf. below), it is possible to activate links that originate from specific objects.

If, for example, a message reception, i. e., method execution join point, is to be decorated instance-locally, this can be done by restricting the corresponding link to that very object. This is possible because a message *reception* hook is local to the object: it lies in the class the object is an instance of.

Should, on the other hand, a message *send* (i. e., a call join point) be decorated instance-locally (i. e., apply only when sent to a given object), selective activation does not suffice. The reason is that the message send hook lies in the class from which the call originates, not in the class of the target instance.

H. Dynamic Deployment Workflow

The hooks inserted at load-time are never withdrawn, but reification can be dynamically (de)activated on a per-link basis. To that end, a *run-time* representation of the link has to be obtained from the link object and made available to the application. Run-time links are instances of classes implementing the `RTLink` interface introduced above.

The `RTLink` interface defines the method `setActiveForLink()`. It accepts an instance of a class implementing the `Active` interface. So, basically, it is possible to dynamically change activation conditions attached to links for the entire link by calling this method. Some other related methods can selectively (de)activate the link for dedicated classes and objects.

There exist predefined `Active` implementations, named `ON` and `OFF`. When they are passed to `setActiveForLink()` and stored in the link, the activation check performed at the hook will yield the respective value and either allow or suppress reification.

Undeployment is triggered by sending the `setActiveForLink()` message with the `OFF` parameter to all links constituting a particular aspect.

I. Other Systems

Iguana/J [131, 132, 83] is another system with powerful reflective capabilities that can be used to implement AOP. The implementors of Iguana/J do however not lay the major accent on the tool's support for AOP; they rather regard it as a general-purpose tool for providing behavioural reflection on the Java platform. It does not bring a plugin interface for AOP languages like Reflex.

With respect to their features, Iguana/J and Reflex do not differ much. The main differences lie in the approach to the definition of reflection protocols, and in the implementation. Reflex is implemented in Java *only* and runs on every JVM that supports custom class loaders. Conversely, Iguana/J brings a protocol definition language of its own. Definitions written in that language must be compiled to Java classes using a dedicated compiler. Moreover, Iguana/J's implementation exploits the low-level JIT interface of the Sun JVM (version 1.3). The plugin that accesses the interface is written in C.

2.10. AspectS

A. Language Presentation

AspectS [82, 20] is an AOP language implemented in the Squeak Smalltalk environment [84, 142]. AspectS is in fact a framework implemented on top of Smalltalk reflection to support AOP. It does not define any new language constructs and all aspects are implemented using only Smalltalk's reflective capabilities.

The AspectS join point model is very simple: only message receptions are supported as join points. This does not restrict expressiveness, since message passing is *the* core mechanism in Smalltalk anyway: method invocations as well as member accesses are implemented using messages.

The pointcut language of AspectS is Smalltalk itself, which allows to use the full power of the Smalltalk language to describe pointcuts. A pointcut in AspectS is a Smalltalk collection that simply enumerates join points.

For the implementation of the sample aspect, a Smalltalk class `HelloWorld` with a method `hello` like the one in Lst. 2.23 is used. The aspect is defined in a class named `HelloAspect`, inherits from `AsAspect` and has one method named `adviceHello`, whose code is shown in Lst. 2.24.

```
1 HelloWorld>>hello
2     Transcript show: 'Hello, world!'
```

Listing 2.23: Application class for the sample aspect in AspectS.

```
1 HelloAspect>>adviceHello
2     ↑ AsBeforeAfterAdvice
3         qualifier: (AsAdviceQualifier attributes: { #receiverClassSpecific. })
4         pointcut: [OrderedCollection
5             with: (AsJoinPointDescriptor
6                 targetClass: HelloWorld targetSelector: #hello)]
7         beforeBlock: [:receiver :arguments :aspect :client |
8             Transcript show: 'I am an advice.']
```

Listing 2.24: Advice definition for the sample aspect in AspectS.

The method defines and returns a so-called before/after advice for which only a before block is given; hence, it establishes a before advice. The advice is configured to affect *all* instances of the `HelloWorld` class in line 3.

The pointcut to whose matching join point shadows the advice is to be attached is set up in lines 4–6. The parameter is simply a collection containing one element, namely a join point descriptor representing receptions of the `HelloWorld>>hello` message. Finally, lines 7–8 contain the definition of advice behaviour.

B. Execution Model Architecture

B.1 In Smalltalk, the border between application and underlying execution environment is blurred. AspectS is implemented as an add-on to the Squeak Smalltalk system *itself*. Various classes have been added to the environment, and moreover, several standard Smalltalk classes were extended to support the AspectS framework. Effectively, this makes AOP mechanisms pervasive in AspectS.

Access to the AOP infrastructure is provided through the AspectS classes (e. g., `AsAspect`, `AsAdvice`, ...). All AspectS classes' names begin with the prefix `As` because Squeak does not have namespaces.

B.2 It is not easy to draw a distinction between compile-time and run-time in Smalltalk, due to the language's integrated approach to software development. Development activities and application executions usually take place in the same virtual machine instance

running an image, and the image contains both the application and the entire Smalltalk development environment. In the context of this work, all AOP-related operations are hence said to be carried out at run-time.

B.3 The join point model, covering only message sends, allows for the only basic weaving technique used by AspectS to be method wrappers. Reflection and meta-programming are crucial and heavily used in the implementation of AspectS.

C. Programming Model Implementation

C.1 Aspects are represented by subclasses of `AsAspect`, and concrete aspects are instances of such aspect classes. The `AsAspect` class defines basic principles, such as a protocol for advice methods, `install`/`uninstall` methods, etc., needed by all aspects. An aspect references its advice, implemented as instances of the `AsAdvice` class.

Methods in instances of `AsAspect` subclasses that define advice functionality begin with `advice`. That way, the AspectS framework can easily recognise such methods. An `advice...` method returns an instance of the `AsAdvice` subclasses.

Advice functionality is actually implemented in closures that are passed at advice instance construction time. These closures are represented as Smalltalk `BlockContext` instances referenced from the `AsAdvice` instance. Moreover, the advice references an `AsAdviceQualifier` defining the advice's applicability. Advice qualifiers will be dealt with in detail below.

An advice block is a closure that, depending on the kind of advice the block is defined for, accepts a certain number of arguments of specific types. Among the arguments are, e.g., the *receiver* of the message constituting the join point, the *arguments* passed along with the message, the *aspect* to which the advice belongs, and the *sender* of the message.

The aforementioned arguments are common for all kinds of advice, and the only ones a *before* advice accepts. An *after* advice additionally accepts a parameter representing the *return value* of the message execution. A *handler* advice takes an additional argument for the *thrown exception*.

An *around* advice takes an additional argument representing the *method* around which the advice is wrapped. This argument is used for proceeding. There is no keyword, but an idiom, for proceeding. It looks as follows:

```
1 clientMethod valueWithReceiver: receiver arguments: arguments
```

The values `receiver` and `arguments` are the first and second parameters of the around block.

C.2 Like most language concepts in Smalltalk, AspectS aspects are also first-class entities. The full reflective power of the Smalltalk environment is usable in conjunction with aspects as well as other entities of the environment.

C.3 There is a certain amount of protocol to which AOP entities in AspectS have to adhere. The elements of the protocol have been mentioned in the above description of the internal representation of aspects, advice, and other concepts. In summary,

- classes representing aspects must inherit from `AsAspect`,
- advice are instances of `AsAdvice` that are returned from methods in the aspect classes whose names begin with `advice...`, and
- the interface of advice is defined by their kind; they accept standardised sets of parameters.

D. Join Point Model Implementation

D.1 The join point model employed in AspectS is very simple. Message sends are the only kind of join point that can be matched. So, the model of the application that is used for join point exposure is the sequence of messages being sent between application objects. Since essentially all objects, including those of the Smalltalk environment, belong to the application, the entire Smalltalk system is exposed to the AOP infrastructure in terms of join points.

D.2 Any join point is represented by a `AsJoinPointDescriptor`. In accordance with the join point model of AspectS, an `AsJoinPointDescriptor` instance solely describes a specific message send by means of a *target class* and *message selector*, each of which is represented by the appropriate Smalltalk entities. Pointcuts are represented as collections of `AsJoinPointDescriptor`s.

This model does not support polymorphic message sends. An `AsJoinPointDescriptor` always describes exactly the class/selector combination it represents; subclasses implementing the same message are not covered. This can, e.g., be met by querying all subclasses of a given class and creating join point descriptors for all of them.

E. Pointcut Model Implementation

Join point shadows are retrieved through reflection on standard Smalltalk meta-objects. This is done when the pointcut-specifying blocks (cf. above) are evaluated. Since all join point shadows are message sends, it is not necessary to browse code in order to evaluate a join point shadow "query".

Statically determinable sets of join point shadows are directly expressed as collections of `AsJoinPointDescriptor` objects. They are assembled programmatically.

Where it cannot be statically determined whether a particular message send is a join point shadow every time it is encountered, advice qualifiers are used to perform dynamic checks. An `AsAdviceQualifier` denotes whether an advice applies to a message send only if, e.g., the message was sent from a specific instance or class, or if the receiver is a specific instance.

An aspect instance references sets to store receivers, senders, and sender classes for which it may be specific. When an appropriate advice qualifier is given, the respective

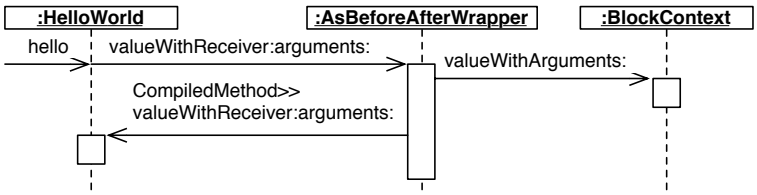

Figure 2.11.: Control flow for the sample advice execution in AspectS.

fields are queried before any advice is executed. The different sets can be arbitrarily modified while the aspect instance exists, thereby leading to a dynamic adaptability of the aspect. Advice qualifiers are also used to specify `cflow` behaviour.

F. Weaving Implementation

F.1 No application *code* is ever altered due to weaving in AspectS. All weaving takes place at the meta-level. Weaving in AspectS means to alter the values that are returned by Smalltalk's original lookup mechanism for late-bound methods. The lookup mechanism is still the same. Yet, the actual return values are different due to method wrappers, who are the core concept of weaving in AspectS. An `AsMethodWrapper` is a direct heir of the Smalltalk class `CompiledMethod`. Thus, a method wrapper is fit to replace a `CompiledMethod` entry in a specific class's method dictionary.

When a message implementation is decorated with advice functionality, its entry in the corresponding class's method dictionary is modified to reference a wrapper. The wrapper invokes advice functionality and yields control to the original implementation. The order in which this happens depends on the nature of the advice.

If several advice are applied to the same message, the wrappers wrap each other. Effectively, this leads to wrapper chains being placed in between the method dictionary and a compiled method.

F.2 In order to execute a `CompiledMethod`, the Smalltalk virtual machine sends it the `valueWithReceiver:arguments:` message. In `CompiledMethod`, this method is implemented such that it sends the `perform:withArguments:` message to the receiver. The `valueWithReceiver:arguments:` message is, in each of the method wrapper classes in AspectS, implemented in a way specific to the requirements of the particular wrapper. For example, the `AsBeforeAfterWrapper`, responsible for representing *before* and *after* advice, first evaluates the before block, then executes the wrapped method, and then evaluates the after block.

This control flow is illustrated in Fig. 2.11. The sequence diagram shows that the message sent to the `HelloWorld` instance is actually received by the `AsBeforeAfterWrapper`, which invokes the before advice (represented as a `BlockContext`). The actual `hello` method is invoked by sending its `CompiledMethod` representation the appropriate message.

F.3 Activation conditions as defined through `AsAdviceQualifier`s lead to residual logic being executed prior to advice bodies. Selective activation is achieved via activation blocks corresponding to advice qualifiers given by the developer. Activation blocks are evaluated prior to a possible advice execution, which occurs or is omitted based on the result.

Activation conditions that need to check for sender instances or classes or even `cflow` reify the call stack and perform a corresponding lookup thereon. The call stack is a first-class construct in Smalltalk systems.

G. Advice Instance Management

G.1 Advice functionality is defined in blocks in `advice...` methods in heirs of the `AsAspect` class. Therefore, `self` in advice blocks always references instances of `AsAspect`. Effectively, the aspect instances themselves are the advice instances.

G.2 Aspects in AspectS can be scoped. Scoping them to single instances is possible by using advice qualifiers. Advice qualifiers allow for making aspects specific to given *sender* or *receiver* instances. That is, a given advice is executed only if the message it is attached to was sent from a specified object, or to a specified object.

Scoping aspects to threads needs further programmatic effort. If advice shall apply only in the context of given threads, the advice have to check for thread identities themselves.

In all of the scoping cases, advice instances are self-managed as in the case of unscoped aspects.

H. Dynamic Deployment Workflow

Aspect deployment, called "installation" in AspectS terminology, is started by sending the `install` message to an instance of a subclass of `AsAspect`. Subsequently, the advice defined by the aspect are collected and the appropriate method wrappers are generated, along with their activation conditions. Finally, the method wrappers are installed in the affected classes' method dictionaries, or, more precisely, in the appropriate places in the corresponding wrapper chains (according to the defined aspect application order). During this process, no code is ever modified; weaving entirely takes place at the meta-level.

Aspect undeployment ("uninstallation") is started by sending `uninstall` to an aspect instance. Basically, aspect undeployment workflow is the reverse of aspect deployment workflow. Aspect uninstallation leads to wrapper uninstallation, which removes the respective wrapper from the wrapper chain. The corresponding method dictionaries are updated accordingly.

I. Other Systems

Apostle [49, 48, 11] is a port of AspectJ 0.8 to Smalltalk. Only an early version of Apostle is available for experiments. Apostle is based on source code modification and

does not exploit Smalltalk's meta-programming capabilities to the degree that AspectS achieves. The driving force behind the development of Apostle was an investigation of incremental weaving.

CARMA [74, 39] is another implementation for AOP in Smalltalk. It also follows a source-code level weaving approach. It is however special in that it utilises PROLOG queries in its pointcut language. Thereby, CARMA supports expressing pointcuts using logic meta-programming; the expressiveness of its pointcut language is very high.

Morphing Aspects

AspectS has dedicated support for a special weaving technique called "morphing aspects" [75]. The idea behind the concept of a morphing aspect is that, when it is deployed, not all of the join point shadows it affects are immediately decorated with residues and advice invocations, but that this is done on the fly as the join points are about to occur. The former weaving approach is called *complete weaving*, while the latter coming in conjunction with morphing aspects is called *continuous weaving*.

The main motivation for continuous weaving is to avoid the overhead induced by residues. This is especially evident when certain join point shadows only very seldom emit an actual join point, but when the residues responsible for determining this are nevertheless executed very frequently.

When a morphing aspect is installed, only a small initial set of join point shadows is immediately decorated with advice. This initial set comprises of those join point shadows that are sure to be reached from the point where aspect installation takes place. Later on, the aspect dynamically adapts the set of join point shadows at which advice invocations are woven; the set may grow or shrink based on the aspect's needs.

To determine the set of affected join point shadows, an aspect's join points are subdivided into *dependent* and *independent* ones. A dependent join point is one that can only be reached when another join point has been reached before. An independent join point does not depend on another one. The corresponding join point shadows are called dependent and independent shadows, respectively. Dependent shadows do not have to be decorated until the shadows they depend on are reached.

In its current implementation, a morphing aspect's continuous weaving behaviour is configured by the programmer. The programmer has to provide information on the initial set of (independent) join points and the continuous weaving process itself, stating which join points lead to the decoration of which dependent join points when reached.

3. Steamloom: A Virtual Machine with Aspect Support

3.1. Introduction

This chapter[1] is devoted to describing the design and implementation of the Steamloom virtual machine. Steamloom is the result of extending a virtual machine for the Java programming language to *natively* support core mechanisms of aspect-oriented programming.

The idea driving the development of Steamloom is that of a generally applicable substrate for aspect-oriented programming languages. It should provide low-level mechanisms specific to the execution models of aspect-oriented programming languages in general rather than just supporting *one* specific such language. The envisioned VM was an execution layer to be targeted by various AOP language compilers. Its model should be extensible to ensure that support for new languages could be added easily.

For the sake of supporting *dynamic* AOP, the VM should lay a complete representation of AOP concepts into the programmer's hands; i. e., all such concepts (aspects, advice, pointcuts, ...) should be available in the form of directly manipulatable entities of the programming model. Access to them should be provided through a comprehensive API, not through language extensions (this also enhances the ability of language compilers to target such a platform).

Avoiding the introduction of new language constructs and instead just providing an API means that crosscutting functionality is, in principle, constituted by "ordinary" classes, objects, and methods. That is, any method should be usable as an advice, and any instance of a class should be usable as an instance to which advice invocations are sent. The API should be used to establish the appropriate composition of such entities at the meta-level.

Furthermore, the execution of crosscutting behaviour—i. e., the execution of residues and invocation of advice—should take place as *implicitly* and *efficiently* as possible. Implicitness means that explicit checks should be avoided wherever tenable, and that no extensive infrastructure should be involved in dispatching advice at join point shadows.

For the time being, the first implementation of these ideas is Steamloom, a virtual machine for AOP-enabled Java. The mechanisms and structures that have been exploited to implement Steamloom can be come upon in any virtual machine with an object-oriented execution model, though. Thus, similar support for other object-oriented execution platforms, such as the .NET CLR, is feasible in principle.

[1] Some of the material presented in this chapter has previously been published [32, 79, 78].

Steamloom was not implemented from scratch. Instead, an existing JVM implementation was chosen as a foundation that was extended and augmented with the appropriate mechanisms. This chapter sets out with a brief overview of the VM implementations that were available at the time the work on Steamloom was begun, and explains the choice for one of them. This one VM, the Jikes RVM by IBM [97], is then described in as much detail as is needed for a thorough understanding of the extensions that were applied in implementing Steamloom on top of it.

The description of the Jikes RVM is followed by detailed presentations of various aspects of the implementation of Steamloom. They focus, in that order, on

- an architectural overview of Steamloom,
- Steamloom's programming model and interface to the AOP functionality,
- the implementations of aspects, pointcuts, and join points,
- bytecode management in the VM,
- the weaver implementation (covering both the generation of woven code and the way it is made to take effect), and
- support Steamloom has for advice instance management, scoped aspects, and dynamic pointcuts.

An evaluation of Steamloom and other approaches will be presented in the next chapter.

3.2. Virtual Machines

Implementing a virtual machine for a complex language from scratch is a demanding task. Since the goal of developing Steamloom was not to implement an entirely new VM for an entirely new language, but to provide execution-layer support for a language extension, it was decided that modifying a pre-existing VM was more auspicious. The prominence of AspectJ, which is an aspect-oriented extension for Java, led to the decision to implement VM support for Java-based AOP.

Prior to setting out with actually implementing Steamloom, a VM to build upon had to be chosen. These most important requirements for the VM to be chosen were:

- *Performance* of the VM had to be good, to be able to compare with approaches based on the standard VM.
- It had to be an *as complete as possible* implementation of Java, to be able to run real applications.
- The VM had to be *well documented*. This requirement strongly supports the following one.
- It had, obviously, to be *easily extensible*.

The VM implementation with the best performance that was available in source code at the time was Sun's reference implementation called HotSpot [143]. While it obviously met, and continues to meet, the first two of the above requirements, it falls short in meeting the latter two. Documentation of the VM internals is scarce. Moreover, it is written in a mix of C, C++, and assembler that makes it hard to grasp.

Support is, on average, good in open-source communities, so it was decided to base the work on Steamloom on an open-source JVM implementation. At the time, there were several open-source JVM projects going on, some of which were briefly evaluated before the decision was made. The open-source virtual machines that were looked at were Kaffe, SableVM and the Jikes Research Virtual Machine (RVM).

Kaffe The Kaffe VM [103] was first implemented as a commercial product, but was released as open source under the GNU General Public License [73] in late 2002. It is implemented in C. When it was made available to the public, it already comprised an interpreter and a JIT compiler. However, its performance at that time was not very good, and the implementation of the Java class libraries it used was quite incomplete[2]. Also, its documentation was in a very rough state. So, it was not an option to use Kaffe in implementing Steamloom.

SableVM Unlike Kaffe, SableVM [68, 67, 136] is a research VM written in C that was originally focused on efficient Java bytecode *interpretation*. Thus, when it was released in 2002, it did not comprise a JIT compiler. Therefore its performance was, albeit very good for an interpreter-based VM, not competitive with regard to, especially, the Jikes RVM that is discussed next.

SableVM relies on GNU Classpath [71], an open-source version of the standard Java class libraries. Unfortunately, Classpath was incomplete in that not all standard classes are fully implemented, e. g., AWT and Swing. However, most Java applications not using a GUI were supported, so the adoption of Classpath was not considered a major drawback.

Jikes RVM Lastly, the Jikes RVM from IBM [6, 4, 7, 8, 97] was taken into account. It is intended as a research platform for virtually all aspects of VM implementations, e. g., garbage collection, JIT compilation, adaptive optimisation and object model implementations.

The Jikes RVM ("Jikes" for short, not to be confused with the Jikes Java compiler [95]) is implemented in Java, but has nevertheless excellent performance characteristics. It is elaborately documented in numerous research publications and in its source code. The developer and Jikes-oriented research communities are very active.

Like SableVM, Jikes relies on the GNU Classpath libraries and thus suffers from its limitations as well. However, Jikes implemented most of the Java language specification [72] apart from small parts of the Java Native Interface (JNI).

[2] In late 2002, the developers of Kaffe switched to the GNU Classpath [71] libraries. Work on Steamloom had then already begun.

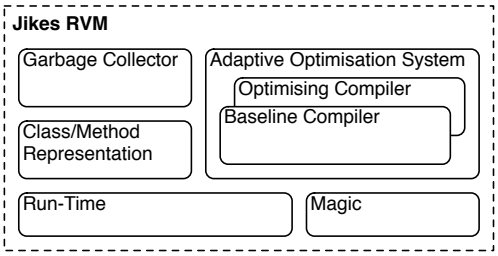

Figure 3.1.: Architectural building blocks of Jikes.

The first three of the four requirements are clearly fulfilled. So is the fourth; but extensibility is further backed by Jikes' implementation language, Java. In fact, developing extensions to Jikes is mitigated because errors in, e. g., pointer arithmetics are not possible. Java's being a strongly-typed language also helps in facilitating an efficient development process.

In the end, it was decided to use Jikes, mostly because its excellent documentation and active support, but also because it was implemented in Java.

3.3. The Jikes Research Virtual Machine

The Jikes Research Virtual Machine (Jikes RVM, or Jikes for short) [6, 4, 7, 8, 97] was originally created by IBM in 1999 under the name "Jalapeño". In 2004, it was released as a SourceForge project. All of the time, Jikes has been available as an open-source project under the Common Public License [44].

The overall openness of Jikes' architecture makes it very valuable for experimental extensions. Jikes is a virtual machine for Java implemented in Java. There is no interpreter; Jikes is completely based on just-in-time compilation, providing an efficient JIT compiler infrastructure with adaptive optimisation of Java methods [13, 16, 15].

The version of Jikes that was used in implementing Steamloom is version 2.3.1. At the time of this writing, version 2.4.0 is available. Steamloom was not adapted to every new release to avoid spending too much time with adapting the extensions to the sometimes subtle changes. All descriptions of Jikes below refer to version 2.3.1.

3.3.1. Overall Architecture and Build Process

As mentioned above, Jikes is, for the most part, written in Java. The standard class library implementation it uses is GNU Classpath [71]; more precisely, Jikes 2.3.1 and Steamloom use Classpath version 0.07. Fig. 3.1 shows the building blocks of Jikes itself; the Classpath libraries are omitted. The various building blocks will be outlined below. In this section, the focus is on describing the build process of Jikes.

First of all, the Jikes sources are compiled to Java bytecode. Basically, this is sufficient to execute the VM running on another one, but to achieve good performance, Jikes is

compiled to machine code using its own infrastructure. This is achieved in a *bootstrapping* process.

A "host" JVM—in the case of Steamloom, the Blackdown 1.4.1 VM [31]—is used to run a Java application called the *boot image writer*. This application basically starts up Jikes and creates a mock-up version [7] of it in memory, which is then written to a file, the so-called *boot image*. To actually run Jikes, a small C application, the *boot image loader*, is used to load the boot image into memory and jump to the VM's boot routine.

The boot image, once constructed, is fixed and cannot be changed. This implies that various configuration choices for the VM have to be made prior to boot image creation, namely at VM compilation time. Such choices are, for example, which garbage collector or JIT compiler infrastructure (for details on these, see the following sections) are to be used. These configuration choices have significant effect on which classes are included in the boot image.

Another important choice is that of the compiler used to generate native code for the boot image from the Jikes bytecodes. Jikes has two compilers; a quick one, called *baseline compiler*, that generates comparatively inefficient code, and a slower one that generates highly optimised code (cf. Sec. 3.3.3). It is possible to combine both of them under the control of the adaptive optimisation system (AOS; cf. Sec. 3.3.4). In that case, methods are initially compiled with the baseline compiler and later optimised, based on execution profiles.

For Jikes, several build models can be chosen, ranging from a "prototype" to a "production" build. In the "prototype" version, the baseline compiler is used for both creating the boot image and compiling methods at run-time, leading to inferior performance but quick VM build cycles. In the "production" build, the optimising compiler is used to generate the boot image, and the AOS is used to manage application method compilation at run-time. This configuration takes very long to build, but produces very good performance when executing Java applications.

3.3.2. Classes, Methods and Objects

Jikes uses a meta-model implementation to represent Java application entities, such as classes, methods, and fields, at run-time. The most important meta-model classes are displayed in Fig. 3.2. All *types* that can occur in the Java programming language are represented by an instance of the appropriate subclass of `VM_Type`. The primitive data types, such as `int`, `double`, or `void`, are represented by instances of `VM_Primitive`. Consequently, arrays are represented by `VM_Array` instances, and for every class loaded in or part of Jikes, there exists an instance of `VM_Class`.

The `VM_Class` instance references the meta-objects representing the class's methods and fields. Every such class member is represented by an instance of a subclass of `VM_Member`. Instances of `VM_Field` stand for fields, and methods are in turn represented by instances of the appropriate `VM_Method` subclass. An instance of `VM_NormalMethod`, for example, contains, among other data, the respective method's bytecode instructions in the form of a `byte` array.

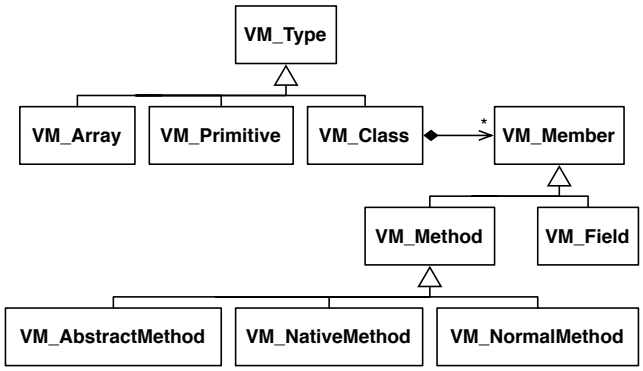

Figure 3.2.: Jikes meta-model classes.

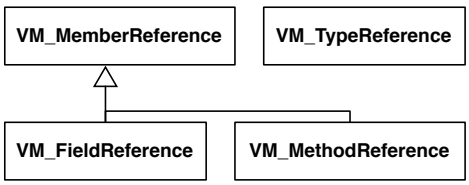

Figure 3.3.: Jikes reference classes.

Altogether, instances of these classes form a complete meta-model of all classes, methods and fields that constitute the VM itself and the application that it executes.

In addition to the classes representing *actual* entities of the meta-model, there exists a collection of *reference* classes as shown in Fig. 3.3. The reference types are used to optimise the class loading and entity resolving process.

For example, whenever a class is loaded, the other classes that are referenced from it are not immediately resolved as well, but references are created for them. These act as proxies until the referenced entity is actually needed: when the `resolve()` method is invoked on a reference object, the referenced class is loaded lazily.

Instances of `VM_TypeReference` act as proxies for as yet unresolved types of all kinds. Correspondingly, `VM_MemberReference`s of the two possible kinds represent unresolved methods and fields.

An *instance* is represented in memory by an object that is a concatenation of slots for header, attributes and other information (cf. Fig. 3.4). The status field is followed by a field holding a reference to the TIB (type information block) of the object, which in turn is followed by the contiguously laid out instance fields.

The TIB is of special interest. It exists once per class and contains all kinds of information related to that class that are of interest while an application is running. The TIB layout is shown in Fig. 3.5. First of all, a TIB is itself an object to the VM: it is treated as an array of `Object`s. Since the TIB is an object, it needs to have a TIB reference on its own; it is that of the `VM_Array` instance representing the `Object[]` array

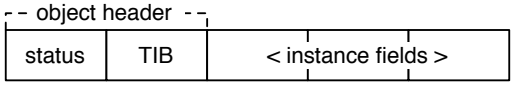

Figure 3.4.: Layout of an object in Jikes (after [7]).

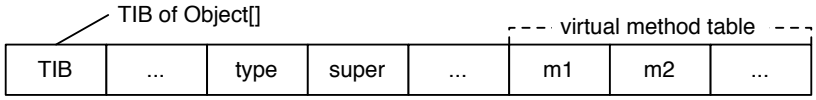

Figure 3.5.: Simplified layout of a type information block (TIB) in Jikes.

type.

The *type* field of the TIB references the type the TIB stands for, i. e., basically, the respective `VM_Type` instance. The *super* field references an array containing the internal IDs of all super classes of this class. This array is maintained for the VM to be able to perform fast subclass checks [5].

The last section of the TIB is very important: it is the virtual method dispatch table for the type represented by the TIB. Each entry of the table points to the machine code of the respective method, or to a stub responsible for dynamic linking of methods upon their first invocation (cf. below).

All static members of classes are held in the JTOC (Jikes table of contents). The JTOC is an array of references to various kinds of data: static class members (fields and methods), numerical and string constants, and also the TIBs of all classes.

3.3.3. Method Compilation

Prior to the first invocation of a method, any TIB and JTOC method entry points to the singleton *lazy compilation stub*, which is a VM-internal Java method, as illustrated by Fig. 3.6 (index 1). The first time a method is invoked, the stub is executed. It inspects the call stack to retrieve the callee object, its class, and the called method. Using this information, the corresponding `VM_NormalMethod`[3] holding the method's bytecode can be retrieved. The stub initiates compilation of the method, sets the respective TIB (or JTOC, in case of a static method) entry to point to the compiled code and executes the method (Fig. 3.6, index 2). Next time the method is called, the compiled code is executed. This technique is called *lazy compilation*.

In the Jikes production build, as was used in implementing Steamloom, this initial compilation step is performed by the *baseline compiler*. The baseline compiler is very straightforward and quick. It simply emulates the semantics of the Java stack machine when compiling bytecodes to native code. It performs no optimisations whatsoever. The code it produces does, naturally, not exhibit high performance.

[3]The treatment of native methods is not discussed here.

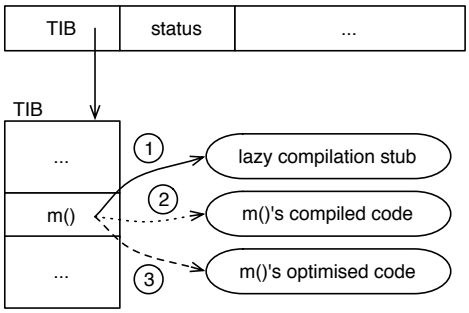

Figure 3.6.: Jikes' normal treatment of methods.

For the generation of fast native code, the *optimising compiler* is used instead. It is, in the production build, triggered by the adaptive optimisation system (cf. below). The optimising compiler operates, at several stages, on several intermediate representations of bytecodes and applies different optimisations at each of the stages [36, 154] (see Fig. 3.7).

Initially, the Java bytecode is transformed to a high-level intermediate representation (HIR). Method inlining is performed during this step. At all stages, from high- over low- to machine-level intermediate representation (LIR, MIR), appropriate optimisations are performed prior to transforming the IR to the next-lower representation. In the end, an instruction array containing the method's optimised machine code is generated.

The optimising compiler offers three levels of optimisation, at each of which a different set of optimisations, increasing in aggressiveness, are applied. The time needed for optimised compilation is certainly higher than that of baseline compilation, and it increases with the level of optimisations.

3.3.4. Adaptive Optimisation

The adaptive optimisation system (AOS) of Jikes has several subsystems [13, 15]. They are organised in several VM threads that each fulfil a specific task related to adaptive optimisation.

The *run-time measurements subsystem* gathers profiling data and stores it in the AOS database. Profiling is done by collecting samples at method epilogues (i. e., when a method returns). From such samples, it can be derived which methods are being executed most often. For methods executed very often, it can be decided to compile them at a higher level of optimisation.

The *controller* receives events from the measurements subsystem and, based on them and stored data, decides on method recompilation. To achieve this, it uses a heuristic [14] to estimate the cost of compiling the method versus the cost of executing it in its current unoptimised (or not-so-optimised) form.

Finally, the *recompilation subsystem* which is notified by the controller whenever a method has to be recompiled, takes care of doing so, invoking the optimising compiler at the appropriate optimisation level. The method is recompiled accordingly and made to

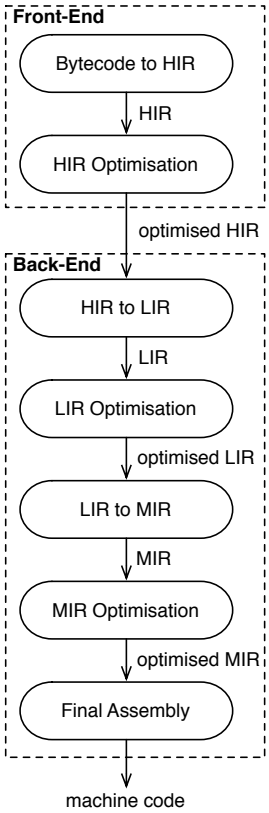

Figure 3.7.: Compilation stages of the Jikes optimising compiler (after [36]).

replace the now obsolete, less optimised version while the application is running (Fig. 3.6, index 3).

Optimised recompilation even works for methods that are currently being executed. A technique called *on-stack replacement* (OSR) [63] is used to accomplish this. OSR logic extracts the current state and program counter from the method call stack frame in question. From the extracted information, a sequence of bytecodes is created that is responsible for re-establishing that very state and then jumping to the instruction address at which the method's execution was interrupted due to the OSR request. This bytecode sequence is prepended to the respective method's bytecodes, which is then recompiled. Execution of the method is then resumed by jumping to the prologue, that reestablishes the correct state and proceeds with the method's execution.

3.3.5. Run-Time Infrastructure

Jikes maintains a number of *virtual processors* that abstract from the actual hardware and operating system resources the VM is running on [7]. Each of the virtual processors is implemented as a single operating system thread.

This abstraction is used to allow for a custom implementation of threads and synchronisation that is independent of operating system specifics. To that end, Jikes provides a thread implementation of its own, in which Java threads are mapped to the virtual processors.

Jikes threads can yield at specified points only. The so-called *yield points* are inserted by the JIT compilers at safe points in methods, namely in method prologues and epilogues, and at back-branches, such as e. g, jumps from the end of a loop body to the loop header. At yield points, the current thread invokes the VM infrastructure that checks whether a thread switch is at hand and performs it if necessary.

In the running VM, a virtual processor is represented by an instance of `VM_Processor`. Such an object maintains the threads attached to it and can be explicitly told to enable or disable thread switching. Threads are represented by instances of `VM_Thread`. A `VM_Thread` instance keeps, among others, the thread's call stack in the form of a `byte` array.

Steamloom is built using a single virtual processor regardless of the capabilities of the underlying hardware. This is due to the fact that the Linux distribution on which Steamloom was developed did not support more than one virtual processor because of the employed version of the C standard library[4].

3.3.6. Magic Code

Even though Jikes is implemented in Java, some parts of its functionality cannot be directly expressed in that language. The most prominent example of such code are methods that access raw memory [7], e. g., to allocate an object or to move it to a new location during garbage collection. Other examples include low-level code for exception handling, thread switching, and locking. To be able to perform such operations transparently by invoking a Java method, the concept of *magic code* was introduced [7].

The `VM_Magic` class contains numerous static methods that each have no proper implementation in source code (they simply throw an exception). Instead, the two compilers of Jikes check, whenever they are about to compile a method invocation, whether the respective method is "magic". If so, they do not generate normal invocation code, but directly output appropriate native or intermediate instructions that perform the desired operation. Thus, magic methods can be regarded as *macros* that are evaluated at compile-time to generate dedicated code that cannot otherwise be expressed in Java language constructs.

3.3.7. Memory Management and Garbage Collection

Memory management in Jikes is handled by MMTk (Memory Management Toolkit) [30]. MMTk is designed for portability and extensibility: it is written in Java and mostly isolated from the VM itself. The interface to the VM is narrow and very clean, which allows its adoption in other virtual machine implementations as well. It also clearly

[4] This setting is recommended in the Jikes User Guide as of version 2.3.1, which comes with that version of the Jikes RVM source code distribution.

separates memory management interfaces from their implementations and thus allows for devising new garbage collectors easily.

Garbage collection is still present in various parts of the VM source code. For example, the baseline compiler computes *garbage collection maps* for that maintain, for each bytecode index at which garbage collection may occur, information on which stack slots contain reference or primitive values. *Most* of the memory management logic, e. g., actual allocation and collection functionality, is however encapsulated in the MMTk packages.

MMTk was originally developed as a part of Jikes but later factored out as a separate project. Nevertheless, Jikes still relies on MMTk and its capabilities. Therefore, Jikes provides a wide choice of garbage collectors, ranging from simple semi-space collectors to advanced implementations, e. g., a generational mark-sweep collector.

The collector that was used in Steamloom is a copying mark-sweep one [99]. Mark-sweep collection is done by first traversing the object graph, *marking* all objects reachable from a set of root objects; and by then iterating over the heap, *sweeping* all unreferenced objects, thereby making the space they occupied available again. A copying mark-sweep collector separates the heap into two spaces: newly created objects are allocated in the so-called *copying* space, while objects that survive the sweep phase are moved to the *non-copying* space. The two spaces are flipped each time collection takes place.

This collector is the one recommended for production builds of Jikes 2.3.1. The memory management policy of this GC separates the heap into distinct spaces, of which the *immortal space* is important for VM-internal data. Objects in immortal space are guaranteedly *never* garbage-collected or moved. Jikes uses immortal space to store objects whose references should never vary, e. g., TIBs.

Due to Jikes' architecture, Java applications that run on Jikes and Jikes itself share the same heap. This means that VM-internal objects (apart from those residing in immortal space) and instances of application classes are not separated in memory. The effect of this is an increased number of garbage collections when applications are run that make extensive use of the heap, which is exhausted earlier in such cases.

3.4. Architectural Overview of Steamloom

This section serves the purpose of giving a high-level overview of the architecture of Steamloom. Special interest is paid to the interference with the Jikes RVM. The discussion below is focused on the building blocks of both Jikes and Steamloom. On the one hand, the modifications applied to Jikes' building blocks are outlined, and on the other, the responsibilities of the major blocks of Steamloom are adumbrated.

Fig. 3.8 shows an overview of the entire system. The parts contributed by Jikes and Steamloom are separated. Those parts influenced or contributed by Steamloom are marked grey.

Steamloom brings three major dowries, namely its *bytecode* and *aspect management*, and its *weaving* component. Albeit connections from these three modules to the Jikes RVM exist, they are mostly isolated from Jikes.

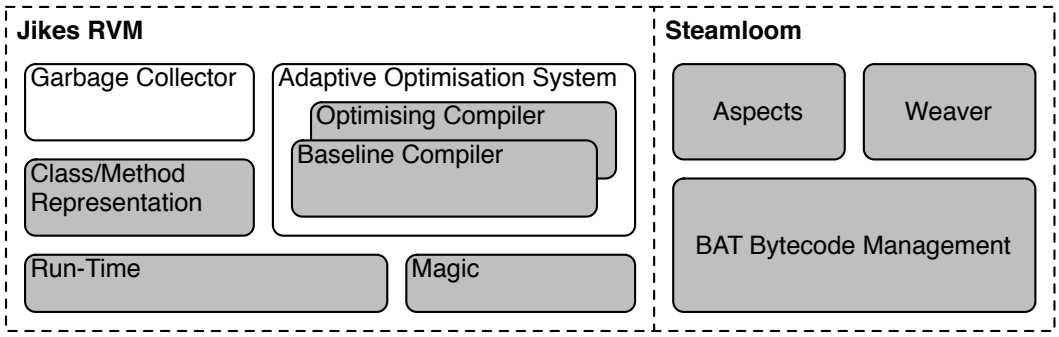

Figure 3.8.: Architectural overview of the Jikes RVM and Steamloom.

The *aspects* module contains classes responsible for the representation of aspects, pointcuts, and advice (cf. Sec. 3.6). The classes in this module are in some cases just data structures that do not contain big fractions of the crucial AOP-related functionality.

However, some classes bear part of the weaving functionality used by the *weaver* module, which is responsible for coordinating the generation of woven code and for its insertion at join point shadows. The weaver has a direct narrow interface to the VM: once method bytecodes have been updated during weaving, it triggers the affected methods' recompilation (cf. Sec. 3.8).

While the two aforementioned Steamloom modules are loosely coupled to the Jikes RVM, the *BAT bytecode management* module has a comparatively high degree of integration with the VM. In Steamloom, BAT is used for representing application methods' bytecode instructions, for retrieving join point shadows, and for weaving aspect functionality into them. BAT is a collection of classes and interfaces that model Java class files and their contents. It defines interfaces to model elements of class file meta-data. Various Jikes classes were modified to implement these interfaces in the course of implementing Steamloom. Also, the bytecode representation for application methods in the VM was replaced with BAT. BAT and its integration with the Jikes RVM are described in detail in Sec. 3.7.

From Fig. 3.8, it can be seen that the garbage collector of the Jikes RVM is its only building block that is not affected by the implementation of Steamloom on top of the VM. Virtually all other parts have been modified or augmented in the course of implementing Steamloom. This points up the high degree of integration that the Steamloom functionality has with that of Jikes.

As written above, the VM's representation of method bytecodes was replaced by the representation adopted by BAT. This has numerous effects on parts of the VM. Of course, the *representations of classes and methods* were modified to implement the interfaces defined by BAT. Also, the VM-internal representation of classes was added data structures for advice instance management (cf. Sec. 3.9).

To avoid a gross modification of the VM's two compilers, it was decided to retain the interface from method representations to the compilers (called the *bytecode stream*).

However, this meant that the implementation of this interface itself had to be reimplemented for processing the BAT bytecode representation instead of the original Jikes RVM one (for details, see Sec. 3.7).

The *compilers* were not left completely unmodified, though. They were augmented to be able to understand several new bytecodes that were added to the VM to support certain aspect-oriented mechanisms. Details on these are presented throughout the following sections.

Another slight extension was made to the *optimising compiler*. After a weaving operation in Steamloom, it might be necessary to recompile not only the method that was subject to weaving, but also those methods where it was inlined during a previous optimised compilation (for details on this, see Sec. 3.8). To ensure that all such methods are recompiled as appropriate, the optimising compiler needs to monitor inlining decisions. This extension was added in implementing Steamloom: now, each method knows where it is inlined.

Other than the two compilers, the *adaptive optimising system* was not modified. Steamloom only exploits its capabilities of storing method call profiles, but the system did not have to be augmented to support Steamloom.

The *run-time* module is responsible for managing execution of applications in the VM. Among these tasks is, for example, thread management and scheduling. The VM's thread representation was altered to support efficient applicability checks for thread-safe advice execution (cf. Sec. 3.10).

Since operations on raw memory and pointers are not possible to implement in Java, *magic code* serves as the interface between the VM and the underlying machine. The magic module defines several methods calls to which are, when met by a compiler, treated like compiler macros that lead to the immediate generation of dedicated machine code fulfilling machine-oriented tasks. In implementing Steamloom, several magic routines were needed to support, e. g., `around` advice (cf. Sec. 3.8).

3.5. Steamloom's Programming Model

In this section, a short introduction to Steamloom's programming model and API will be given by means of three simple examples. The first example is an implementation of the sample aspect from Ch. 2 in Steamloom, the second example deals with logging of member accesses, and the third with caching computation results.

Sample Aspect The sample aspect is implemented as follows. The advice method is implemented in a class `HelloAdvice`, whose source code is shown in Lst. 3.1. The aspect itself is set up and deployed in the hello world application class, whose augmented source code is shown in Lst. 3.2. The listing has been slightly simplified; exception handling code has been removed.

In the `setupAspect()` method in lines 5–14, the aspect is set up and deployed as follows:

```
1  public class HelloAdvice {
2      public void advice() {
3          System.out.println("I am an advice.");
4      }
5  }
```

Listing 3.1: Advice class for the sample aspect in Steamloom.

```
1  public class HelloWorld {
2      static {
3          setupAspect();
4      }
5      public static void setupAspect() {
6          Method m = HelloAdvice.class.getDeclaredMethod("advice", null);
7          BeforeAdvice ad = new BeforeAdvice(m, new HelloAdvice());
8          PointcutDesignator p = SimpleParser.getPointcut(
9                  "call(void HelloWorld.hello())"
10             );
11         Aspect a = new Aspect();
12         a.associate(ad, p);
13         a.deploy();
14     }
15     public static void main(String[] args) {
16         HelloWorld h = new HelloWorld();
17         h.hello();
18     }
19     public void hello() {
20         System.out.println("Hello, world!");
21     }
22 }
```

Listing 3.2: The hello world application with sample aspect in Steamloom.

1. In lines 6–7, the advice is created. The method `HelloAdvice.advice()` is to be used as the advice method in this example. An ordinary reflection object for this method is created. After that, an instance of the `BeforeAdvice` class is created from the advice method and an instance of the `HelloAdvice` class. The meaning of this creation is that whenever the advice is invoked, the `advice()` method on the instance passed to the `BeforeAdvice` constructor is to be called.

2. The pointcut with which the advice is to be associated is created in lines 8–10. A `PointcutDesignator` is created through the simple AspectJ syntax parser. The `PointcutDesignator` object hierarchy (cf. 3.6.2) could also be assembled by hand using API calls, but using a parser is more convenient.

3. The aspect is created and deployed in lines 11–13. An instance of `Aspect` is created, and the created advice is associated with the pointcut. Finally, the aspect is deployed.

The `setupAspect()` method is invoked from the static initialiser of the `HelloWorld` class, so that it is already deployed when the `main()` method is entered. This is necessary

because a running method cannot deploy aspects that affect the running method itself (cf. 3.8.8).

It should be noted that the parser provided as part of Steamloom does not support those elements of the AspectJ syntax that deal with making join point context information available to advice, and those that deal with addressing pointcuts by name. Join point context access is specified in conjunction with defining advice (cf. below). Pointcuts can be referenced from within other pointcuts when they are assembled using the Steamloom API directly. The parser does support wildcards, though.

Setter Logging For the second example, a collection of classes modelling shapes will be used, and a small drawing application that creates three shapes and moves them around by random values ten times. The source code of the `figures` package is given in Lst. 3.3, that of the `Display` class in Lst. 3.4, and that of the drawing application itself in Lst. 3.5.

Steamloom is used to implement an aspect that logs all setting operations on the `int` members of instances of `Point`. The aspect will moreover output the respective `Point` object. The code that achieves this is shown in Lst. 3.6.

The most interesting part is the method `setupAspect()`. It assembles and returns an `Aspect` instance with the desired behaviour. The aspect is assembled using basically the same mechanisms as the previous sample aspect. However, this time, the advice method accepts a parameter, namely the `Point` object whose field is set.

Parameter passing to the advice is configured in line 8. In conjunction with a field set operation, the join point shadow's *target* object is the object whose field is going to be set. So, the advice needs to be passed the target of the field set operation, which is achieved by invoking `appendTarget()` on the advice.

Result Caching The third example implements a cache to speed up the recursive computation of Fibonacci numbers. The application in Lst. 3.7 shows the source code of the `Fib` class that only defines the recursive `fib()` method. Lst. 3.8 shows the source code of a cache for computation results: they are stored in an automatically expanded array. A value of -1 indicates that no result has yet been stored at a given index.

In Lst. 3.9, the source code of a class is shown that, when the static method `deploy()` is invoked on it, sets up and deploys an aspect that caches Fibonacci computation results to avoid repetitive deep recursion.

The `deploy()` method in lines 13–25 sets up and deploys an aspect that wraps each call to `Fib.fib()` in an around advice, which is implemented in the method `CachingAspect.cacheFib()`. This method accepts an `AroundClosure` parameter which represents the join point context of the advice invocation (cf. Sec. 3.6.4 for details), so the `AroundAdvice` is set up to pass this parameter by invoking `appendClosure()` on it.

In the around advice method (lines 3–12), the `AroundClosure` is downcast to a `CallClosure`, which represents the join point context at a method call. From the closure, it retrieves the argument `k` passed to the call and checks whether the result of `fib(k)` has already been cached. If so, the cached result is immediately returned from

```
1  package figures;
2
3  public interface Shape {
4    public void moveBy(int dx, int dy);
5    public void draw();
6  }
7
8  public class Point implements Shape {
9    private int x, y;
10
11   public Point(int x, int y) { this.x = x; this.y = y; }
12
13   public void setX(int x) { this.x = x; }
14   public void setY(int y) { this.y = y; }
15   public int getX() { return x; }
16   public int getY() { return y; }
17
18   public void moveBy(int dx, int dy) { x += dx; y += dy; }
19
20   public void draw() { System.out.println("(" + x + "," + y + ")"); }
21   public String toString() { return super.toString() + "(" + x + "," + y + ")"; }
22 }
23
24 public class Line implements Shape {
25   private Point a, b;
26
27   public Line(Point a, Point b) { this.a = a; this.b = b; }
28
29   public void moveBy(int dx, int dy) { a.moveBy(dx, dy); b.moveBy(dx, dy); }
30
31   public void draw() {
32     System.out.println("Line[(" + a.getX() + "," + a.getY() + "),("
33                                 + b.getX() + "," + b.getY() + ")]");
34   }
35
36   public String toString() {
37     return super.toString() + "Line[(" + a.getX() + "," + a.getY() + "),("
38                                        + b.getX() + "," + b.getY() + ")]";
39   }
40 }
```

Listing 3.3: Source code of the `figures` package.

the advice.

Otherwise, the result is computed by actually proceeding with the invocation of the `fib()` method (line 9). The `proceed()` method *always* returns an `Object` and performs auto-boxing for primitive values. The result of the computation is stored in the cache and returned.

3.6. The Steamloom Aspect Model

In this section, a brief overview of Steamloom's implementation of the five core concepts at the heart of AOP, namely *aspects*, *pointcuts*, *join points*, *join point shadows*, and *advice* is given. In the following, the focus is not on the details of functionality implementation,

```
1  package display;
2
3  public class Display {
4    private static Display instance = new Display();
5
6    public static Display instance() { return instance; }
7
8    Set shapes = new HashSet();
9
10   public void update() {
11     for(Iterator it = shapes.iterator(); it.hasNext(); )
12       ((Shape) it.next()).draw();
13   }
14
15   public void addShape(Shape s) { shapes.add(s); }
16 }
```

Listing 3.4: Source code of the `Display` class.

```
1  package simple;
2
3  public class SimpleDraw {
4    Display d;
5    Shape[] s;
6    Random r = new Random();
7
8    public SimpleDraw() {
9      s = new Shape[3];
10     s[0] = new Point(10, 10);
11     s[1] = new Point(5, 5);
12     s[2] = new Line(new Point(1, 9), new Point(9, 1));
13     for (int i = 0; i < s.length; i++)
14       Display.instance().addShape(s[i]);
15     Display.instance().update();
16   }
17
18   private int rnd() { return r.nextInt(10) - 5; }
19
20   private void randomMove() { s[r.nextInt(s.length)].moveBy(rnd(), rnd()); }
21
22   public void runDraw() {
23     for (int i = 0; i < 10; i++)
24       randomMove();
25   }
26 }
```

Listing 3.5: Simple drawing application.

but rather on the concepts' representation as *data structures* in Steamloom.

3.6.1. Aspects

Steamloom aspects are first-class entities. They can be created, assembled, deployed and undeployed entirely at run-time. For the high-level structure of the aspect model, see Fig. 3.9. Aspects are represented by instances of the `Aspect` class.

```
public class SetterLoggingReify {

  public static Aspect setupAspect() throws Exception {
    Method logMethod = SetterLoggingReify.class.getDeclaredMethod(
        "log", new Class[] { Point.class }
      );
    Advice logAdvice = new AfterAdvice(logMethod, null);
    logAdvice.appendTarget();
    PointcutDesignator logPcd = SimpleParser.getPointcut(
        "set(int figures.Point.*)"
      );
    Aspect a = new Aspect();
    a.associate(logAdvice, logPcd);
    return a;
  }

  public static void log(Point p) {
    System.out.println("set Point property for " + p.toString());
  }

  public static void main(String[] args) {
    Aspect a = setupAspect();
    a.deploy();
    SimpleDraw sd = new SimpleDraw();
    sd.runDraw();
  }
}
```

Listing 3.6: Code to realise a setter logging aspect for the drawing application with Steamloom.

```
public class Fib {
    public int fib(int k) {
        if(k <= 2)
            return 1;
        return fib(k-1) + fib(k-1);
    }
}
```

Listing 3.7: A Fibonacci application.

To Steamloom, an aspect is a mere container mapping pointcuts to advice. Each mapping associates one pointcut with one advice. Such a mapping is represented by an instance of the `AspectUnit` class. An aspect may consist of numerous aspect units, effectively allowing n-to-n relationships between pointcuts and advice.

3.6.2. Pointcuts

A pointcut is represented as a tree of objects, each of which is an instance of a subclass of `PointcutDesignator`. The `PointcutDesignator` class itself is abstract, but its various subclasses can be used to assemble complex pointcuts. The `PointcutDesignator`

```
public class Cache {
    private static int[] cache;
    public static int INITIAL_SIZE = 10;
    static {
        cache = new int[INITIAL_SIZE];
        Arrays.fill(cache, -1);
    }
    private static void expandCache() {
        int[] newCache = new int[cache.length*2];
        System.arraycopy(cache, 0, newCache, 0, cache.length);
        Arrays.fill(newCache, cache.length, newCache.length, -1);
        cache = newCache;
    }
    public static int valueFor(int k) {
        while(k >= cache.length)
            expandCache();
        return cache[k];
    }
    public static void store(int k, int res) {
        while(k >= cache.length)
            expandCache();
        cache[k] = res;
    }
}
```

Listing 3.8: A class for caching computation results.

```
public class CachingAspect {
    Cache cache = new Cache();
    public int cacheFib(AroundClosure c) {
        CallClosure cc = (CallClosure) c;
        int k = cc.getIntArgumentAt(0);
        int res = FibCache.valueFor(k);
        if(res != -1)
            return res;
        res = ((Integer) cc.proceed()).intValue();
        FibCache.store(k, res);
        return res;
    }
    public static void deploy() throws Exception {
        Method mCache = CachingAspect.class.getDeclaredMethod(
                "cacheFib", new Class[] { AroundClosure.class }
            );
        Advice aCache = new AroundAdvice(mCache, new CachingAspect());
        aCache.appendClosure();
        PointcutDesignator pCache = SimpleParser.getPointcut(
                "call(int Fib.fib(int))"
            );
        Aspect a = new Aspect();
        a.associate(aCache, pCache);
        a.deploy();
    }
}
```

Listing 3.9: Fibonacci caching functionality implemented as a Steamloom aspect.

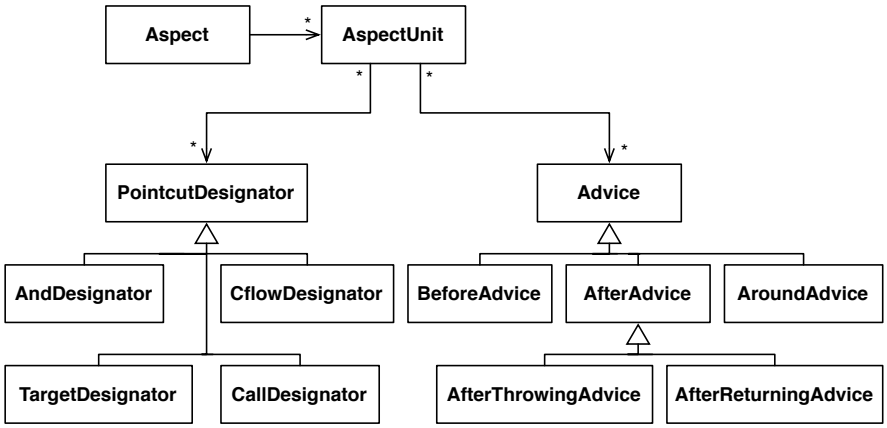

Figure 3.9.: Steamloom model classes.

subclasses[5] model basic designators, like `call`, `set`, `cflow`, `target`, etc., as well as compositional designators that can be used to establish complex pointcuts through logical combinations like `&&`, `||`, and `!`.

Most `PointcutDesignator`s are parameterised with additional information. For example, a `CallDesignator` (representing a `call` pointcut expression) is parameterised with a `MethodPattern`. The entire structure of the pointcut representation mostly adheres to the AspectJ definitions.

As an example, Fig. 3.10 shows the `PointcutDesignator` object tree representing the pointcut expression `call(public void A.m()) || call(public void B.p(A))` applicable to the source code in Lst. 3.10.

```
1  class A {
2    public void m() { ... }
3  }
4
5  class B {
6    A a;
7    public void p(A a) {
8      this.a = a;
9      a.m(); // join point shadow: invokevirtual instruction
10   }
11 }
12
13 class C {
14   B b;
15   public void q() {
16     b.p(new A()); // join point shadow: invokevirtual instruction
17   }
18 }
```

Listing 3.10: Simple example classes.

[5]Not all of the subclasses are shown in the figure.

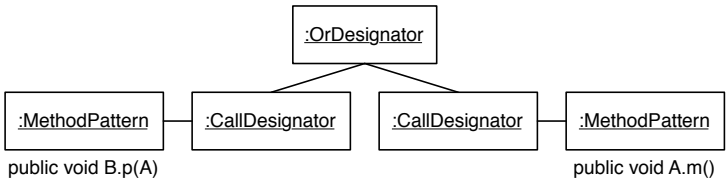

Figure 3.10.: `PointcutDesignator` hierarchy for a pointcut.

Pointcuts can, on the one hand, be assembled programmatically, by directly using the constructors of the various `PointcutDesignator` classes and passing them the appropriate parameters. On the other hand, it is possible to provide parsers that accept a textual representation of a pointcut in a given pointcut language and produce a `PointcutDesignator` object tree therefrom.

While a parser for AspectJ pointcuts is already included in Steamloom, it is possible to develop new parsers for arbitrary languages. Since it is also possible to extend the `PointcutDesignator` hierarchy with entirely new types of pointcuts, the coexistence of multiple pointcut languages is feasible in Steamloom.

3.6.3. Join Point Shadows

Join points have no direct representation since they are, as opposed to meta-level approaches like Reflex (cf. Sec. 2.9), not explicitly reified in Steamloom. Nevertheless, join point *shadows*, as the actual locations in code where join points occur or may occur, have a representation as first-class entities. This is possible because each join point shadow is represented as one or more bytecode instructions; and method bytecodes are, in Steamloom, represented as linked lists of `Instruction` objects, due to the adoption of the BAT bytecode framework (cf. Sec. 3.7). Thus, a join point shadow, i. e., all locations in code where a join point possibly occurs, is represented as a collection of `Instruction` objects.

3.6.4. Advice

Steamloom advice are represented by instances of `Advice` subclasses. For each type of advice, there exists one such class. An `Advice` instance contains a reference to an ordinary Java reflection `Method` object that represents the method providing advice functionality. Additionally, since advice can be passed parameters, the `Advice` instance maintains information on this.

When an advice shall be passed parameters from the context of the join point it advises, the according `append...` methods have to be invoked on the `Advice` object for configuration. If the advice method has n formal parameters, then *exactly* as many calls to an `append...` method are allowed. The `append...` calls must occur in the order of the parameters: the first call specifies which value is passed as the first parameter, and so forth.

There are various methods for appending a parameter to the advice method call:

- `appendAccessibleObject()`: this method can be used when static join point context information is to be passed to the advice. If the advice in question is attached to a field access join point shadow, the parameter passed to it according to `appendAccessibleObject()` will be a `java.lang.reflect.Field` instance representing the respective field. Correspondingly, a `Method` will be passed for method call and execution join point shadows. `AccessibleObject` is the superclass of these two classes, hence the name of the `append...` method.
- `appendArg()`: when an argument to the join point shadow shall be passed to the advice, this method is used. The method accepts an `int` argument that denotes which of the arguments is to be passed. The index for the first argument is 1. It is also possible to pass negative values, which indicates that the argument list is looked at from its end: -1 means the last argument. The `appendArg()` method does not only work for method call or execution, but also for field set join points. In that case, the value passed to the advice is the value stored in the field.
- `appendClosure()`: this method is only used in conjunction with around advice. Its usage is described below.
- `appendException()` and `appendReturnValue()`: these methods only work with after throwing and after returning advice, respectively. They indicate that the thrown exception or value returned from an operation are to be passed to the advice. The `appendReturnValue()` method not only works for results returned from methods, but also for results from field get operations. The correct use of the two methods with respect to the kind of advice they are attached to is enforced, e. g., sending `appendException()` to an `AfterReturningAdvice` yields an exception.
- `appendTarget()`: the target object of the join point at which the advice method is invoked will be passed. This depends on the nature of the join point: at a `call` join point, the method call target object will be passed. Conversely, at a `get` or `set` join point, the instance whose field is being accessed will be passed. Note that `appendTarget()` behaves just like `appendThis()` when used with method execution join points.
- `appendThis()`: whatever object is `this` in the context of the advice method invocation will be passed as a parameter.

Around Advice

Unlike before and after advice methods, methods constituting functionality of around advice must adhere to a protocol. This is because an around advice is executed *in place* of a join point shadow, without any guarantee that the replaced shadow is ever executed. Therefore, the method invoked as an around advice must fulfil the following requirements:

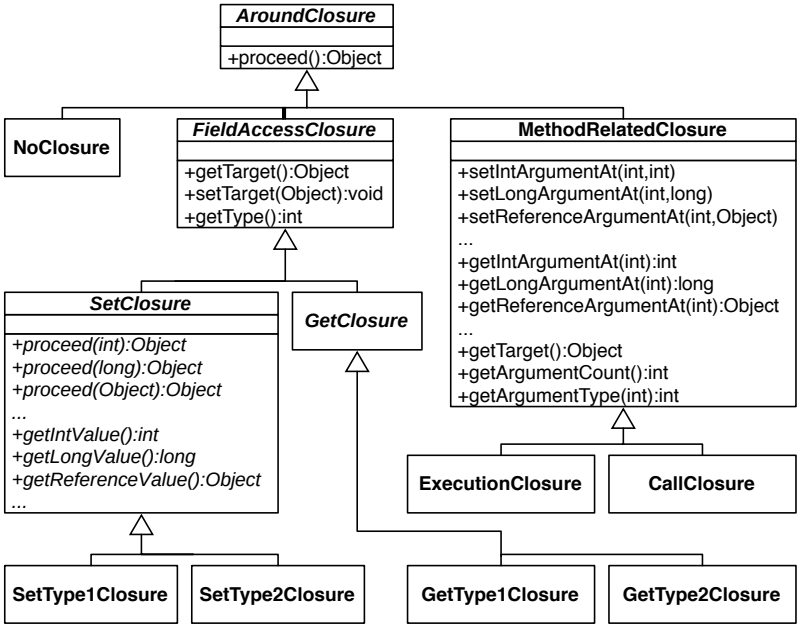

Figure 3.11.: Around closure hierarchy in Steamloom.

- The advice method *must* have the same or a more specialised return type as the result type of the join point shadow it replaces (`void` for a field set shadow).
- If `proceed()` is used within the advice, the *first* parameter of the advice method *must* be of the type `AroundClosure`. This is done by invoking `appendClosure()` on the `AroundAdvice` instance before *any other* parameters are defined. If `proceed()` is not used, parameters may be arbitrary, as for before and after advice methods.

This protocol is enforced by the Steamloom infrastructure to the degree that the return type of the around advice method is checked for conformance. Moreover, the `appendClosure()` call is ensured to append the *first* parameter. The other protocol elements must be obeyed by the programmer.

The `AroundClosure` instance is created transparently by the infrastructure at runtime and passed to the advice method. The closure is of interest only if an around advice method has to proceed with the execution of the original join point shadow, or if it wants to access information from the join point context.

The `AroundClosure` class is an abstract superclass of specialised closure classes for special types of join point shadows (see Fig. 3.11). There exist closure types for the two kinds of field accesses and for method calls and executions. The closures for field accesses are further specialised for types entities of which consume one or two words in memory. A generalised closure exists for method call and execution join points.

Access to Join Point Context The subclasses of `AroundClosure` provide various methods to access—i. e., read and write—elements of the join point's context.

It is important to note that, in conjunction with around advice, it is *not* possible to use the context reification methods introduced above. Whenever an around advice wants to reify join point context information, it *must* be passed an `AroundClosure`, which *must* be declared to be the first parameter to the advice method using `appendClosure()`.

Since the different subclasses of `AroundClosure` being specialised for the kind of join point shadow they represent, have different interfaces, a compiler targeting Steamloom must generate a downcast to the respective subclass. Sensible types to cast to are `SetClosure`, `GetClosure` and `CallClosure`. The other types in Fig. 3.11 are for internal purposes. The `SetType1Closure`. `SetType2Closure`, `GetType1Closure` and `GetType2Closure` classes are responsible for representing field accesses to fields storing values of one and two words length, respectively. The `NoClosure` class will be explained in Sec. 3.8.6.

For example, `GetClosure` and `SetClosure` instances allow for reifying the target object of the assignment. `SetClosure`s moreover provide for retrieving the value that is about to be stored. Both closure types allow for retrieving the type of the field that is about to be set.

Analogously, the `CallClosure` and `ExecutionClosure` provide access to the call target and parameters of the method invocation. The various `get<Type>ArgumentAt(int)` methods accept an `int` parameter that denotes the index of the parameter in the method signature. A value of zero indicates the first formal parameter for all kinds—virtual and static—of method calls. The call target can be reified by invoking `getTarget()`, or `getThis()` in `ExecutionClosure`s.

In all of these cases, it is the responsibility of the compiler generating code for Steamloom to generate an invocation of the correct method.

So far, only read access to the join point context has been discussed. Write access is discussed in the immediately following paragraphs about proceeding from around advice.

Proceeding If an advice method has to proceed with the original join point shadow, this can be achieved by simply invoking `proceed()` on the `AroundClosure` instance that has been passed to the advice method. This method can be invoked without any parameters, in which case execution proceeds with no further side effect; i. e., parameters passed to the join point shadow are not modified in any way. If proceeding should go with modified attributes, e. g., a different value for a field set, or different parameters for a method call, there are certain ways to modify join point context information.

For field set join points, encapsulated in a `SetClosure`, the compiler targeting Steamloom can choose between two ways to pass another value than the original one to the field. On the one hand, it is possible to directly invoke the appropriate `proceed()` method with the desired value. On the other hand, the value can be explicitly set by invoking one of the `set<Type>Value()` methods and calling `proceed()` after that.

It is, for field get and set join points, even possible to alter the *target* of the field access operation. That is, the field read or written is still the same, but belongs to another instance. This is possible by invoking `setTarget()` on the closure, but it should be used with extreme care.

A call join point shadow has only one implementation of `proceed()`, but parameters to the call may be modified by invoking the appropriate `set<Type>Argument()` methods. These methods each accept an index and the value to be passed instead of the original one. The index is zero-based and denotes at which position in the call arguments the new value is to be inserted. The compiler targeting Steamloom must ensure that no erroneous operations are performed. For example, it is technically possible to modify the call target by invoking `setTarget()`, but doing so may lead to unforeseeable results.

Return Values The `proceed()` method *always* returns an `Object`. This value is `null` for field set join point shadows. For field gets and method calls, it contains the retrieved value, or the method's return value, respectively.

If the join point shadow's result type is a scalar, the implementation of `proceed()` performs auto-boxing, returning an appropriate instance of, e. g., `Integer` or `Long`. Again, the compiler generating code for Steamloom has to generate a correct downcast and code for the extraction of the value from the container instance.

Complicated `proceed()` API The protocol around the `proceed()` method may seem overly complicated when compared to AspectJ's straightforward syntax. It has to be noted, however, that AspectJ's `proceed` keyword is a pseudo function that is translated into the appropriate invocations by the AspectJ compiler. Steamloom, on the other hand, does not provide any extensions at language level, but offers support for AOP solely through an API. Hence, the capabilities AspectJ provides so easily must be provided through static interfaces in Steamloom, which leads to a comparatively complicated protocol.

Other approaches that do not extend the programming language itself follow similar approaches. In AspectWerkz (cf. Sec. 2.6) and Spring AOP (cf. Sec. 2.8, around advice also accept a closure-like parameter to which a `proceed()` message can be sent.

3.7. Bytecode Management in Steamloom

In this section, first the bytecode toolkit BAT [24], used by Steamloom for join point shadow retrieval and weaving, is introduced. After that, details on the integration of BAT into Jikes are provided.

3.7.1. BAT: Bytecode Augmentation Toolkit

BAT (Bytecode Augmentation Toolkit) [24] is an open-source bytecode engineering toolkit released under a BSD style license [25]. It was designed from the start with the intention of allowing for easy integration with other tools and frameworks and even Java virtual machines.

The version of BAT that is employed in Steamloom is not the one available from the web page [24]. It is rather a branch of the original BAT project that was adapted to fit the needs of integration into Jikes. BAT has undergone extensive further development

since. All descriptions of BAT internals that are presented here refer to the version that was used to build Steamloom.

Unlike other bytecode toolkits [26, 96], BAT declares interfaces for all of the Java language's meta-model entities in accordance with the JVM Specification [109]. The overall design of BAT aims at decoupling interfaces from implementations, so every class file element is at first only modelled as an interface. Of course, BAT provides standard implementations for all elements.

BAT offers functionality to change existing bytecode and to create completely new sequences of bytecode instructions. In BAT, all information stored in a class file is made available by a fine-grained object hierarchy, the meta-model entities, down to the level of an instruction, i. e., every bytecode instruction is represented as an object. A method's instruction sequence is stored as a doubly-linked list making updates of bytecode sequences efficient.

An important feature which makes BAT particularly well suited as part of an AOP-enabling infrastructure is its framework for efficient and fine-grained localisation of join point shadows, the so-called *bytecode pointcut framework*. Filter objects can be composed to describe the join point shadows to be selected by their properties. Such properties can be, for example, various elements of the signature of a method call/execution, or a field access join point shadow's field name, etc. The bytecode instructions that pass the filter are returned when the filter is applied to a class file representation. In addition, there are filters that represent the logical "`&&`", "`||`" and "`!`" operators, which can be used to build more complex filters. For illustration, regard the following filter that selects all instructions that access the field `out` declared in the class `java.lang.System`:

```
1 Filter f = new AndFilter(
2   new FieldAccessNameFilter("out"),
3   new FieldAccessDeclaringClassFilter("java.lang.System")
4 );
```

BAT analyses composite filter objects and optimises them to reduce the number of steps required when applying the filter to a class file.

Internally, filters use *matchers*. These are, in principle, also filters but do not collect instruction objects. Their purpose is to match types and identifiers against type and identifier patterns. Hence, they play a crucial role in resolving expressions containing wildcards.

3.7.2. Integration of BAT into Jikes

BAT was integrated into the Jikes RVM to entirely replace the VM's bytecode management for application classes. This was done for three reasons. Firstly, BAT provides sophisticated mechanisms for pointcut evaluation and join point shadow retrieval in the form of filters. Secondly, BAT allows for easily modifying method bytecode instructions during weaving because of its representation of method bytecodes as linked lists of instruction objects. Thirdly, a core idea of the integrative approach of Steamloom is to let AOP functionality and the run-time environment work on the same representation of the application to avoid conversions between internal and external formats.

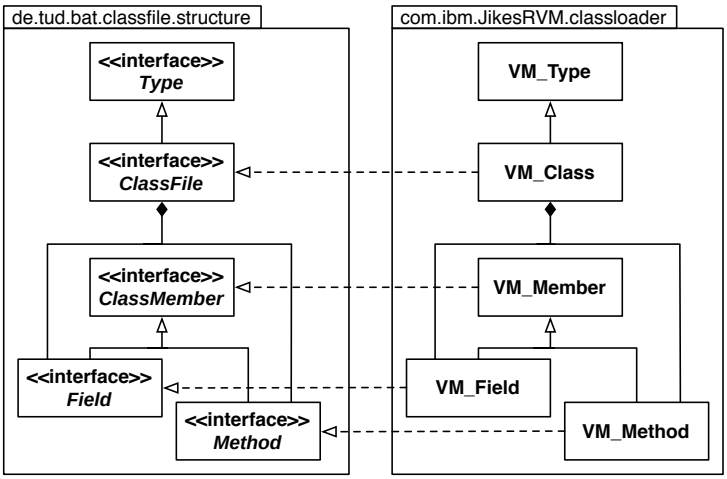

Figure 3.12.: Parallel inheritance hierarchies in BAT and Jikes.

Within Steamloom, BAT takes over the complete bytecode instruction management for application classes, thereby facilitating retrieval of join point shadows, and enabling to manipulate the bytecode instructions which is not possible in Jikes. Some of the meta-model classes provided by Jikes have been modified to implement their respective BAT counterpart interfaces: that way, Jikes entities become instances of the BAT meta-model, and BAT can access them when evaluating filters.

Mapping Meta-Models

An excerpt of the relationships of the BAT and Jikes meta-models is displayed in Fig. 3.12. It is interesting to see how both BAT and Jikes implement the class file structure in similar ways. Both tools certainly adhere to the class file structure as defined in the JVM specification [109]. Moreover, there are similarities in the way they are modelled, which is a clear advantage for the task of integrating them with each other.

The various Jikes reference classes (cf. Sec. 3.3.2, Fig. 3.3) were not modified to implement the corresponding BAT interfaces. Instead, the object adapter pattern [69] was applied to them, so that each Jikes reference instance references a respective BAT object. The reason for adopting this approach is that, for BATified entities, both versions of the references are needed: references from the BAT library and, VM-internal references. They are created using different reference factories. Also, the BAT reference classes provide functionality that is needed within BAT, but cannot integrated with Jikes references using inheritance since the VM's reference classes have their own inheritance hierarchy. Altogether, this suggested to use an adapter instead of interface implementation.

Retaining Compiler Support

In Jikes, a method's bytecode instructions are represented as `byte` arrays. This is not ideal for AOP support, since weaving frequently encompasses adding instructions to existing code: arrays cannot easily be expanded. BAT uses doubly-linked lists of instruction objects, which is more flexible and allows for manipulating the instructions easily.

The process that transforms bytecode instructions from the fixed array representation into the flexible representation is called *BATification*. The BATification process is carried out at class loading time, but only for application classes; the VM's functionality or classes from the run-time library are not allowed to be decorated with aspects. The decision behind this is that aspects are only to be used for interfering with the application. Of course, *calls into* the aforementioned classes can still be dealt with in pointcut expressions and decorated with advice.

To allow its subsystems, e. g., the JIT compiler, access to bytecode instructions, Jikes provides an abstraction layer, the so-called *bytecode stream*. Bytecode streams are used not only for compilation, but also to compute garbage collection maps [7]. So, method bytecodes are an important source of information for various parts of the VM, and the interface those parts use is always the bytecode stream, implemented in the `VM_BytecodeStream` class.

`VM_BytecodeStream` provides an abstraction from the bytecode instructions of a Java method. The stream can be queried for opcodes and their parameters. For the parameters, there is some type checking to ensure no wrong values are requested.

In the original implementation of `VM_BytecodeStream` in Jikes, the stream iterates over an array of `byte` values, the raw bytecode attribute of the method in question. In Steamloom, method bytecodes are only represented as raw `byte` arrays when the methods have not been BATified; i. e., when they are no application methods.

To keep the interface of `VM_BytecodeStream` while letting the stream iterate not only over `byte` arrays, but also over BAT `Instruction` objects, Jikes' `VM_BytecodeStream` class was made abstract, and several concrete subclasses were provided. Thus, for non-application methods, Jikes' original implementation is used, which was renamed to `VM_SystemBytecodeStream` and made an heir of the now abstract `VM_BytecodeStream`. The class hierarchy is displayed in Fig. 3.13.

For BATified methods, Steamloom provides several alternative implementations of the bytecode stream which also inherit from `VM_BytecodeStream` but iterate over BAT instruction lists instead of `byte` arrays. Their commonalities are implemented in the abstract class `BATBytecodeStream`.

The reason for there being three distinct concrete bytecode stream classes in Steamloom lies in the requirements of on-stack replacement (OSR) logic. OSR, as described in Sec. 3.3.4, prepends a prologue to a method's bytecode instructions that is about to be recompiled for on-stack replacement. During OSR compilation, the compiler at times requests single portions of the complete bytecodes to be represented by a bytecode stream instance: the prologue only, the actual method instructions only, or the entire so-called *synthesised* instructions. Jikes solves this by passing tailored `byte` arrays to

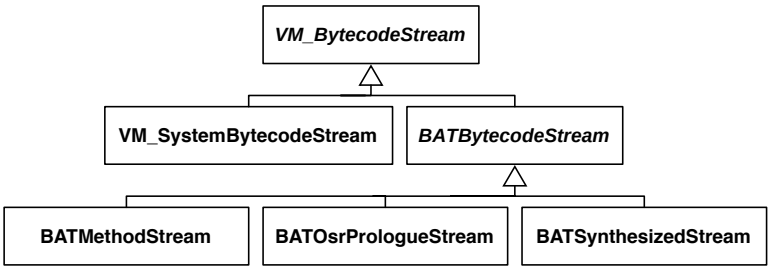

Figure 3.13.: Bytecode stream hierarchy in Steamloom.

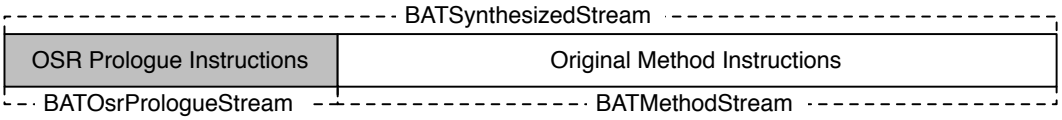

Figure 3.14.: Bytecode ranges over which Steamloom bytecode streams iterate.

the `VM_BytecodeStream` constructor.

Since, in Steamloom, the instruction list cannot be torn apart, tailoring the bytecode stream is achieved by providing the aforementioned three dedicated bytecode stream classes that each iterate over a different range of the instruction list belonging to the method in question (cf. Fig. 3.14).

The differences of the altogether four bytecode stream implementations are hidden by their common interface, thus dependent subsystems of the VM can access method instructions regardless of them being BATified or not.

3.8. Dynamic Aspect Deployment

The deployment process for a given aspect—represented by an instance of the `Aspect` class—is initiated by sending the `deploy()` message to that object. The outline of the ensuing process is as follows:

1. The aspect has been set up with one or more calls to its `associate()` method, binding `PointcutDesignator`s to `Advice`. Each such association is represented by an `AspectUnit` instance in the aspect. The aspect's `deploy()` method iterates over all of its `AspectUnit`s and sends them the `deploy()` message.

2. The aspect unit registers itself with the `AspectUnitRegistry`. The latter splits the (possibly composite) pointcut contained in the aspect unit into basic parts (cf. 3.8.1). To each such part of the split pointcut, the advice is to be attached. An instance of `DeployedAspectUnit` representing the part as a *deployed* entity is created.

3. The join point shadows pertaining to each particular part are retrieved (cf. 3.8.2).

4. The set of methods containing join point shadows is determined, and the weaver (cf. Sec. 3.8.3) is invoked for each of the methods.

5. The weaver modifies the code of the method according to the kind of join point shadow, advice, and possible residues to be attached to the shadow (cf. Sec. 3.8.4). When it is finished with the method, it is invalidated and thus scheduled for recompilation (cf. Sec. 3.8.8).

3.8.1. Pointcut Splitting

Splitting of a composite `PointcutDesignator` yields a collection of pointcuts that, were they linked with an `||` operator, would be semantically equivalent to the original pointcut. Each of the single pointcuts can be composite, but the only composition operator occurring in them is `&&`. An basic pointcut part consists of *one* `PointcutDesignator` selecting an actual *operation* (i. e., a `call`, `execution`, etc. designator), and no or more designators that impose restrictions on the former (e. g., `this`, `target`, `cflow`, ...).

For example, the following pointcut expression

```
1 (call(void X.m()) || execution(void Y.n())) && this(B) && target(Z)
```

would be split into these two elements:

```
1 call(void X.m()) && this(B) && target(Z)
2 execution(void Y.n()) && this(B) && target(Z)
```

The splitting process is implemented in two classes that both act as visitors on `PointcutDesignator` objects: `PointcutSplitter` and `PointcutNormalization`.

The latter serves the purpose of normalising logic expressions in the pointcut. Normalisation in this context means that no composite negated expressions occur in the entire pointcut, but that negations are *only* attached to the leaves of the pointcut designator tree. To achieve this, the normaliser transforms the pointcut designator tree and pushes negations down. When normalisation is done, the splitter also transforms the tree by factoring out all disjunctions.

Pointcuts in this form are easier to be dealt with by the weaver for two reasons. On the one hand, the weaver can deal with the shadows of *one* statically resolvable pointcut at a time and generate residues for them. On the other hand, when negations are attached to atomic conditions, i. e., single pointcut designators, the weaver also just needs to generate a negation for a particular check it weaves.

In a second step of the splitting process, the `DynamicPointcutSplitter` visitor is applied to the parts that the first step has yielded. The purpose of the second step is to build hierarchies of `DeployedAspectUnit` instances, each of which logically represents a piece of woven code that can be treated by the weaver in isolation. For example, a designator like `this(B)` is mappable to a piece of code that checks whether `this` is an instance of the class `B`.

In the context of the above example, the expression

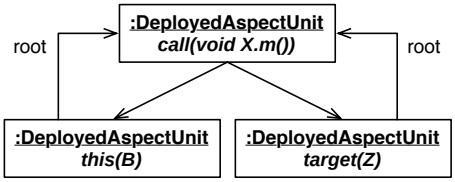

Figure 3.15.: A hierarchy of deployed aspect units.

```
1 call(void X.m()) && this(B) && target(Z)
```

would be transformed to a hierarchy of `DeployedAspectUnit`s like the one shown in Fig. 3.15. Each of the particular nodes knows the root of the hierarchy that it is attached to.

3.8.2. Join Point Shadow Retrieval

Join point shadow retrieval is, in Steamloom, the process of collecting and returning all `Instruction` objects that (may) match a given pointcut. The `PointcutDesignator` class has a method `getJoinPointShadows()` that is called to trigger the retrieval process.

In that method, first a query is created that is subsequently applied to the loaded classes. It is an instance of the BAT `Query` class, created by the `PointcutQueryMapper`, an instance of which visits the elements of the pointcut designator tree and creates the corresponding query elements. For the most join point types, standard queries are created that revert to using BAT filters (cf. Sec. 3.7.1). When such a query is evaluated, it forwards evaluation to the filter it wraps.

For the retrieval of method call and field access join points, filters are not an ideal approach. While a method execution join point shadow can normally easily be retrieved by matching class names against potentially included wildcards and then directly pointing at the method in question, join points of the aforementioned kind are more difficult. This is because shadows of such join points may occur virtually everywhere in the loaded classes.

A field may be accessed in more than one method even if it is declared `private`. All methods that possibly access it have to be scanned by the filters. If a field is declared `protected` or `public`, the number of methods to be scanned increases dramatically. This also holds for method calls, which may occur in arbitrary methods all over the application.

When weaving is taken out at run-time, the speed at which it is performed is eminently important. Join point shadow retrieval may introduce a bottleneck when it is implemented in an inefficient way. As seen above, static analysis based on accessibility properties of methods and fields is not sufficient to speed up join point shadow retrieval. It only reduces the amount of classes and methods that have to be scanned for join point shadows, but the complexity of that process still depends on the number of bytecode instructions loaded—for a public field access or method call pointcut designator to be

resolved to its join point shadows, *all* of them must be scanned.

To solve this problem, Steamloom supports the BAT filters with *indices* that store, per method or field, which methods call or access it. The BAT reference classes for fields and methods have been added dedicated sets that store references to those instructions representing access or call join point shadows for a given field or method.

The functionality for updating indices is implemented in the `Indexer` class. This class implements a callback interface of the Jikes RVM specific to class loading, and its singleton instance registers with the VM. Thus, whenever a class is instantiated (i. e., when all of its members have been loaded, but the initialiser has not yet been run), the indexer is triggered. It iterates over all the class's methods and scans them for field access and method invocation instructions. Once such an instruction is found, it is added to the index of the respective field or method reference.

The indices are stored in the *references* rather in the classes actually representing fields and methods (`VM_Field` and `VM_Method`) because instances of the latter are not created until the class declaring the field or method is actually *resolved*, i. e., finally and completely loaded.

Since the indices *directly* store join point shadows in the form of `Instruction` objects, join point shadow retrieval for field access and method call pointcuts is as simple as retrieving the set from the index. This is a very cheap operation, no more than a field access. It is however possible that pointcuts are further restricted, e. g., by `within` designators. In these cases, BAT filters are applied to the set of already resolved join point shadows to shrink it to only those shadows that actually match the pointcut.

The `PointcutQueryMapper` creates special query objects for method call and field access pointcuts that look up the join point shadows in the indices associated with the method and field references in question.

Evaluation of the `Query` yields a list of all those join point shadows that match or may match the pointcut. They are already filtered with regard to the kind of advice that is going to be attached to them: e. g., for a method execution pointcut to whose join point shadows an after returning advice is going to be attached, all returning instructions are contained in the query result, but not the first instruction of the method. The latter would be contained in the result if a before advice was going to be attached.

From the single instruction objects returned from the query, `JoinPointShadow` instances are assembled. A `JoinPointShadow` contains the instruction objects belonging to the shadow, the statically resolvable part of the according pointcut, the advice to be attached to the shadow, and the dynamically resolvable parts (if any) of the pointcut, i. e., designators like `this` etc. that lead to the generation of residues.

This step is necessary because some kinds of join point shadows—e. g., a method execution join point shadow to which an after returning advice is to be attached—may consist of more than one instruction. Join point retrieval finally returns a list of `JoinPointShadow` instances.

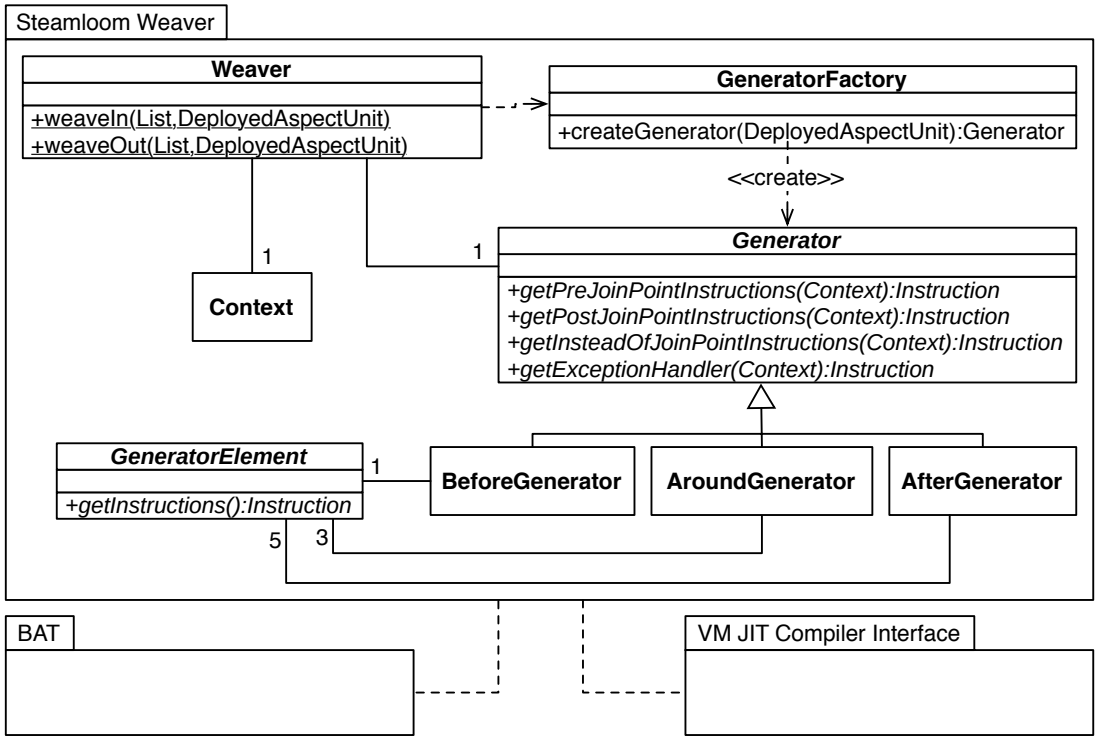

Figure 3.16.: Overview of the Steamloom weaver classes.

3.8.3. The Steamloom Weaver

Once join point shadows for a pointcut are found, resulting in a list of `JoinPointShadow` objects, the advice associated with the pointcut at hand is woven in at the shadow, in a way depending on the advice type. Steamloom does so by using BAT to insert the according instructions.

These instructions are generated by the *weaver* module of Steamloom. An overview of the most important classes contributing to this module is given in Fig. 3.16. The weaver module has been designed and developed with the idea of being reusable. Therefore, the weaver has only two major dependencies: BAT, for actual weaving and generation of woven instructions, and the VM's JIT compiler interface, to trigger recompilation of affected methods. The weaver module is quite tightly coupled to BAT, since it relies on BAT's representation of method bytecodes throughout.

The `Weaver` class is the sole entry point to the weaver's functionality. It exhibits two methods, one for weaving aspect code into application code, and one for removing it. Both accept a list of `JoinPointShadow` instances and a `DeployedAspectUnit`. In the following, it will be described how the weaver acts when the `weaveIn()` method is invoked. The functionality of the `weaveOut()` method will not be described in detail; it is to simply remove the pieces of code that were inserted by `weaveIn()`.

It is crucial for the entire VM to avoid interferences of its JIT compilers and the weaver.

If the AOS decides to recompile a method that is currently being instrumented by the weaver, the compiler accesses inconsistent bytecode instructions, which is extremely unsafe. Hence, the weaver and all compilers of the VM synchronise on the method object they process. That way, a method can *never* be subject to weaving and JIT compilation at the same time.

The weaver generates a `Context` and a `Generator`. The `Context` encapsulates information on how the weaving process modifies the number of local variables in the method subject to weaving. It may be necessary to add new local variables to the method because values need to be stored temporarily.

There are three subclasses of the abstract `Generator` class, one for each type of advice. `Generator`s have four important methods in their interfaces. The method `getPreJoinPointInstructions()` generates instructions that are to be inserted *before* the join point shadow for which woven code is generated. Correspondingly, the method `getPostJoinPointInstructions()` generates instructions that are to be inserted *after* the shadow. The `getInsteadOfJoinPointInstructions()` method is used for around advice: code generated for them *replaces* the original join point shadow. In addition, it may be that weaving code into a method requires an exception handler to be added to that method, which is generated by `getExceptionHandler()`.

The different generator methods are applicable for different advice types as follows:

- For before advice, naturally, only `getPreJoinPointInstructions()` plays a role at all.
- For after advice, both `getPre...` and `getPostJoinPointInstructions()` are relevant. For example, it may be that an after advice attached to a method call is to be passed the target of that call. In that case, the target object must be stored in a local variable *before* the call, for which operation `getPreJoinPointInstructions()` generates code.
- Code for around advice is generated by `getInsteadOfJoinPointInstructions()` if no additional tests are applied to the advice invocation (cf. below).

The `GeneratorFactory` is responsible for creating the appropriate kind of `Generator` according to the advice type contained in the `DeployedAspectUnit`.

All generator classes rely on `GeneratorElement`s during code generation. These are the actual code generators of the weaver. They may be complex, composed of several other elements. There exist concrete element classes for the generation of the following kinds of woven code:

- *advice blocks*, i. e., blocks of woven code,
- *advice method invocations*,
- *tests*, i. e., residues of all kinds, and
- *context access*.

In addition to these, there are several optimising generators that are used to *remove* tests that are known to always fail or pass and, instructions depending on such tests. For example, tests based on the `args` pointcut designator that can mostly be statically evaluated based on the method signatures on hand.

Fig. 3.16 shows that `BeforeGenerator` references only one `GeneratorElement`, while `AfterGenerator` and `AroundGenerator` reference more. The latter two classes require to use several generator elements because they possibly need to generate code to be woven in more than one place; recall the above example of the after advice attached to a call that is passed the call target.

The `AroundGenerator` may have to generate code in several places when the invocation of an around advice is conditional. In the *unconditional* case, it simply replaces the original join point shadow. However, when the advice invocation depends on the result of some residual tests, code must be inserted before and after the original shadow (cf. Sec. 3.8.6 for details).

After the context and generator have been created, the `weaveIn()` method iterates over all `JoinPointShadow` instances that were passed to it, sets up the generation context and applies the created generator to it. The resulting instructions are immediately inserted into the instruction list.

Once the weaver is done with the method, i. e., once the iteration over all of the join point shadows has finished, the method is updated and scheduled for recompilation. This is explained in detail below, in Sec. 3.8.8.

3.8.4. The Shape of Woven Code

Code that is woven in at join point shadows is organised in so-called *advice blocks*. Lst. 3.11 shows the general form of such a block. An advice block is surrounded by two instructions, which are specifically introduced by Steamloom, namely the `beginadvice` and `endadvice` instructions. They contain an identifier of the aspect they belong to and mark the code block containing the advice method call and, if necessary, residues and instructions to prepare the stack for parameter passing and clean-up, i. e., removal of unused return values.

Steamloom introduces several new bytecode instructions: `beginadvice` and `endadvice` are two of them, and more will be introduced below and in Sec. 3.8.7. These special-purpose bytecodes do not have to be generated by any compiler; they are for sole use by the Steamloom infrastructure when it generates dynamically woven code. They are therefore not expected to be present in any class file that is loaded into the VM. The bytecode verifier may safely treat them as illegal when it comes across them during class loading.

The `beginadvice` and `endadvice` instructions serve to facilitate undeployment of specific aspects without reweaving all other aspects that affect a method. When an aspect is undeployed, the corresponding advice block is simply removed from the instruction list of an affected method.

Both instructions are treated like `nop` instructions by the compiler (which effectively means they are ignored) and thus introduce no execution overhead. They basically

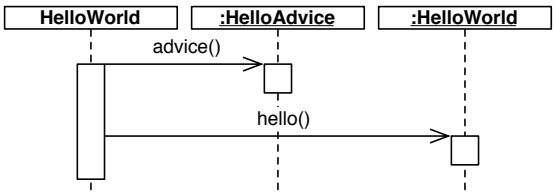

Figure 3.17.: Control flow for the sample aspect in Steamloom.

just serve as tags. However, the `endadvice` has an additional use as a jump target for the residues: failing residual checks are implemented to jump to the corresponding `endadvice` instruction instead of the next actual instruction. This allows for an easy implementation of such dynamic checks, because the advice block can be seen in isolation, and the following instructions need not be regarded by the weaver.

```
1 beginadvice      <advice-id>  ;; mark the beginning of this advice block
2 <residues>                    ;; residual logic (only if required)
3 aaitpush         <ait-index>  ;; push advice instance on stack (if any)
4 <parameters>                  ;; push advice parameters on stack (if needed)
5 invoke...advice  <method>     ;; invoke the advice method
6 <clean-up>                    ;; clean up the stack (if needed)
7 endadvice        <advice-id>  ;; mark the end of this advice block
```

Listing 3.11: General form of an advice block woven in by Steamloom.

Advice are, as mentioned above, normal Java methods. In case an advice method is virtual, advice invocations are sent to objects, so-called *advice instances*. They are directly associated with the classes they advise through *advice instance tables* [76] (cf. Sec. 3.9), an efficient concept for storing advice instances. For their lookup, another specific bytecode instruction, `aaitpush`, was added to the VM that retrieves an advice instance from the table in minimal time. Details on this bytecode instruction are given in Sec. 3.9.

3.8.5. Weaving Before and After Advice

The code responsible for before advice invocations is inserted immediately before the join point shadow it is attached to. For method calls and field accesses, this means that the code is inserted before the respective instruction. For method executions, the advice block is prepended to the first instruction of the method. Fig. 3.17 shows the sequence diagram representing the control flow associated with the sample advice introduced in Ch. 2 when it is used in Steamloom.

After advice are more complicated to deal with. Only in very simple cases can the code generated for an after advice invocation merely be inserted *after* a single instruction. This holds, for example, for after returning advice that are attached to method calls and field accesses and do *not* access join point context. It also holds for after returning advice without context access that are attached to method executions when the respective method has only one exit point. In the latter case, after advice blocks are inserted

immediately before the `return` instruction. If an after returning advice is attached to a method execution that has multiple return instructions, then the advice block is inserted before *each* of them.

An after throwing advice is, for method calls and field accesses, implemented by adding an exception handler to the method containing the join point shadow. This handler wraps the shadow, and its catch block is the advice block. An after throwing advice attached to a method execution join point shadow is implemented by adding an exception handler that wraps the *entire* method. The catch block, again, is the advice block. In both of these cases, the advice block must also contain an `athrow` instruction to pass on the exception.

Join point context access adds more complexity to after advice weaving. Most of a join point's context information is available on the stack only *prior* to the join point's execution. If an after advice intends to access such context, it must be saved in local variables. If an after advice of whatever kind accepts parameters from the join point context, the weaver generates appropriate instructions that save the context elements. These instructions are inserted *before* the join point shadow in question. In Sec. 3.8.7, details on this will be provided.

3.8.6. Weaving Around Advice

For implementing around advice, there are basically two approaches. On the one hand, it is possible to directly inline the around advice code at join point shadows (which AspectJ usually does, cf. Sec. 2.2). On the other hand, closures can be used. Steamloom follows a closure approach. Inlining is, if many methods are affected by an around advice, very expensive to achieve. Moreover, it is overly complicated in dynamic weaving environments, where it is possible that around advice are woven in and out.

Around Advice Blocks The shape of around advice blocks strongly depends on whether the advice invocation depends on the results of residues attached to it. If there are residues, it is possible that the original join point shadow is not entirely replaced by the around advice, but that it is invoked when the tests evaluate to `false`.

If there are no residues, the around advice block is straightforward to generate; it simply replaces the original shadow. If residues are present, the original shadow must not be removed from the bytecode instruction list. Instead, it serves as a jump target for the case that residues fail. Another jump target must be inserted *after* the original shadow; this target is jumped to when the around advice has been executed.

Around advice blocks (the following descriptions focus on around advice blocks without residues) generally are a little different from before and after advice blocks. An example is shown in Lst. 3.12. Most of the elements seen in the previous listing are present, apart from the "clean-up" step. The advice method invocation is assumed to leave the stack in *exactly* the state that the original join point shadow instruction would have left it in. Thus, discarding the return value of the around advice method, as it is done for before and after advice, is not desired.

```
1 beginadvice      <advice-id>  ;; mark the beginning of this advice block
2 <residues>                    ;; residual logic (only if required)
3 aaitpush         <ait-index>  ;; push advice instance on stack (if any)
4 <closure>                     ;; generate AroundClosure (if needed)
5 <parameters>                  ;; push advice parameters on stack (if needed)
6 invoke...advice  <method>     ;; invoke the advice method
7 endadvice        <advice-id>  ;; mark the end of this advice block
```

Listing 3.12: Shape of an around advice block.

Nevertheless, there is a new step, named "closure" in Lst. 3.12. During this step, the `AroundClosure` instance is created and populated with the correct contextual information.

It is possible that the replaced original instruction is later executed, due to a `proceed()` invocation on the closure instance. To be able to perform the original instruction correctly, the closure must know the stack state at the position of that instruction. For that purpose, the state is extracted from the stack and stored in the closure to be reestablished when the around advice proceeds. In case an around advice method is not passed a closure parameter, which indicates that it will neither access join point context information nor proceed, the stack is cleaned of all state that would otherwise have been used as input to the original join point shadow.

The respective `AroundClosure` subclasses provide means for generating the appropriate bytecode instructions that manipulate the stack state accordingly. During weaving, the `getClosureCompositionInstructions()` method of the appropriate closure class is called when code for an around advice invocation is created, and the returned instructions are inserted into the generated woven code.

Closures are created and populated with context information per join point occurrence. For population, the different closure classes have a number of dedicated methods in their interfaces that are subsequently invoked for each context item (such as target objects and arguments). Internally, the closures provide storage for context information.

Around Execution Advice When an around advice is attached to a method execution, the code for invoking it theoretically has to replace the entire method body. Approaches operating at compile-time and load-time to prepare classes for weaving address this by adding a new method to the respective class and moving the body of the advised method to that new method. In a dynamic weaving environment, it is not easily feasible to add new methods to a class that has already been loaded.

Hence, around execution advice are, in Steamloom, implemented as *around call* advice. This is established by converting execution pointcut designators to which around advice are to be attached to call pointcut designators prior to join point shadow retrieval. Subsequently, the join point shadow to be decorated with the around advice is a call shadow, which consists of one instruction and can be dealt with using the mechanisms described above.

Semantically, there are no difficulties with this because the difference between a call and an execution join point is, basically, the identity of the object in which the join point

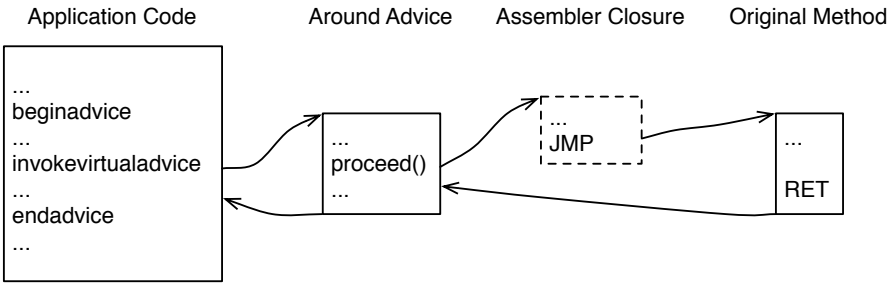

Figure 3.18.: Control flow for around advice execution at method call join points.

occurs. At a call join point, `target` and `this` are (usually) different. At an execution join point, they are commonly regarded to be the same. The implementation of the `ExecutionClosure` accommodates this by treating the retrieval operations for these two in a unified way.

This way of dealing with around execution advice is also not harmful with respect to recursion. Consider, for example, the `fib()` method in Lst. 3.7 on page 110, calls to which are decorated with around advice to perform result caching. In case the advice had been applied to *executions* of the `fib()` method, the weaving result would have been the same as it now is in the example (cf. Sec. 3.5).

The Implementation of `proceed()` The `proceed()` method of the `AroundClosure` subclasses consists of a certain amount of code performing type checking, autoboxing and stack set-up that is gathered around the invocation of the join point shadow surrounded by the around advice in question. This invocation logic looks different for each particular `AroundClosure` type.

Field accesses are implemented in a quasi-reflective manner, exploiting the VM's own capabilities of issuing field get and set operations. The VM provides a set of magic methods (cf. 3.3.6) that perform a given field access operation when passed an object reference and a field offset. The implementations of `proceed()` in the subclasses of `FieldAccessClosure` simply invoke the correct `VM_Magic` method depending on the type of the field in question.

When an around advice proceeds at a method call, the case is different. The control flow executed in this case is outlined in Fig. 3.18. A short array of assembly code is generated that basically executes a reflective call to the method whose invocation is to be triggered by proceeding. The implementation of `CallClosure.proceed()` invokes that very block of machine code. It contains instructions to place the parameters for the method call, which it extracts from the `CallClosure`, on the stack and to eventually invoke the original method. The assembler closure performs an immediate jump to the beginning of the method wrapped by the around advice, so that the wrapped method's returning instruction skips the assembler closure and directly returns to the around advice body.

These assembler closures exist once per wrapped method and are created lazily during

`AroundClosure` creation. They are stored in a repository so that they do not have to be created whenever a `CallClosure` or `ExecutionClosure` is created.

It has been mentioned above that this approach of implementing `proceed()` is quasi-reflective. It is however located at a much deeper level of abstraction than the normal Java reflection API classes are. The VM's mechanisms are directly exploited, without the amount of infrastructure that would be needed, was the standard reflection API employed instead.

The around advice implementation is, at the time of this writing, restricted in three ways:

1. It is only supported by the baseline compiler. The magic code for the optimising compiler has not been implemented due to time constraints. Because of this, the `proceed()` methods are marked to never be optimised. This restriction obviously leads to a performance penalty for around advice, which will be observed in the evaluation in Sec. 4.2.3.

2. Around advice cannot be safely attached to join point shadows that throw exceptions. This is due to the lack of support for optimised code: the assembler closure is not properly represented in an optimised stack frame, which hinders call stack construction for exceptions. For baseline-compiled code, around advice at such join point shadows work.

3. It is not possible to attach more than one around advice to a particular join point shadow. This is because the assembler closure taking over the execution of the wrapped join point shadow is not recognised as a join point shadow itself.

3.8.7. Join Point Context Access and Residues

As mentioned in Sec. 3.6.4, an advice can get access to dynamic context (`this`, target objects, arguments, return values and exceptions) and static context (the accessed field or called method). Moreover, around advice can access a closure object representing the join point they wrap. The code that is generated for around closure access has been described above.

An advice is set up for parameter passing by invoking one or more of the `append...` methods on it, in the order that the parameters are to be passed. Internally, the advice stores a series of parameter descriptors in that very order. The weaver—rather, the appropriate `Generator`—accesses these descriptors and generates an according sequence of bytecode instructions that, all in all, establish the stack state the advice requires.

Dynamic Context Access to dynamic context—i. e., to values available from the stack or local variables—is simply implemented using plain bytecode instructions. If `this` is to be passed to an advice, an `aload_0` instruction is all that is needed. The case is more complex for call targets and arguments, because they are normally not directly accessible as a local variable or lie on top of the stack.

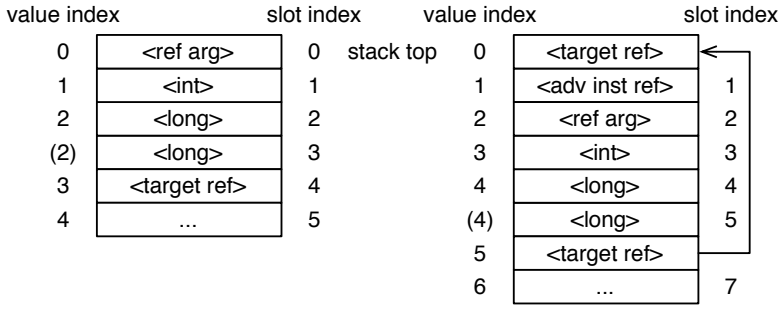

Figure 3.19.: Stack states prior to a method call, and for an advice invocation.

On the left-hand side of Fig. 3.19, a possible state of the stack immediately prior to a call to a virtual method is shown. The method accepts—in the given order—a `long`, an `int`, and a reference parameter. The last parameter lies on top of the stack (slot index 0), the target object of the call lies at slot index 4.

Assume a before advice is attached to the call that shall be passed the target. On the right-hand side of Fig. 3.19, the state of the stack that must be established is shown. The advice method is itself virtual, so an advice instance will be pushed. The target object of the advised method call is not immediately reachable, but must be pushed on top of the stack.

Weaving logic could generate a sequence of instructions that stores all values that are in the way in local variables, pushes them again, and finally pushes a copy of the desired object. The number of bytecode instructions needed for this depends linearly on the depth of the stack at the given point in time. There are two downsides to this approach: on the one hand, the weaver has to generate many instructions, and on the other hand, these instructions, or their corresponding machine code, must be executed later on, whenever the advice is to be invoked.

In Steamloom, this is overcome by means of a new bytecode instruction: `peek`. The `peek` instruction accepts one parameter, namely the index of a slot or value in the stack, and copies that value to the stack top. Note that `peek`'s argument is polymorphic: it can denote both a *slot* and *value* index. These indices differ when values consuming two slots, i. e., `long` or `double` values, are on the stack.

The semantics of an argument to `peek` are set when the bytecode instruction object is created; there exist appropriate constructors of the corresponding instruction class. When value indices are used, the BAT may need to perform a control-flow analysis during JIT compilation to determine the stack layout at the point where the `peek` is met. This slows down compilation. When slot indices are used, this is not necessary, but the stack layout must be known at the time the `peek` instruction is created.

In Steamloom, `peek` is *always* used in contexts where the stack layout is precisely known. When, for example, context access code is to be woven that accesses arguments or targets for a method call, the signature of the method to be called is known. From the signature, the stack layout can be easily deduced. Hence, Steamloom applies the faster slot index semantics of `peek`.

Following the style alleged by the JVM specification [109], two bytecode instructions, `peek` and `peek2` would have been introduced. They both would have accepted *slot* rather than value indices, and the latter would have pushed *two* slots on the stack. However, BAT allows for expressing abstractions like the one implemented in `peek`, and it is more convenient to have a bytecode instruction class whose instances can be easily instructed to behave in one of the two ways.

The VM's compilers were appropriately extended to generate machine or high-level intermediate code from the `peek` instruction. Since `peek` bytecodes are used only in contexts where the stack layout is known or easily deducible from method signatures, the compilers can easily compute the correct slot indices to generate correct code.

Using `peek`, context access is very straightforward to implement. Target objects and arguments can, with a single bytecode instruction, be made available to advice invocations.

Static Context Access to static context is implemented in the following way. Steamloom maintains an array of `AccessibleObject` instances. Each particular join point shadow that needs to access static context is registered with a fixed index, and the object—method or field—in question is, at weave-time, stored in that array. The weaver generates an array access operation with the fixed index that pushes the object on the stack at run-time, so that it can be passed to the advice.

Following this approach avoids the use of actual reflection, which would be expensive. At weave-time, the objects to be passed to the advice are precisely known. So, Steamloom basically follows the same approach that a compile-time weaver would follow to make static context available to advice.

Residues There are two kinds of residues that are identified by the complexity of executing them. Checks for thread-locality—a residue important for Steamloom's thread-local aspects—and `cflow` are relatively complex. They will be dealt with in Secs. 3.10 and 3.11.

The other, less complex category of residues is effected by the employment of pointcut designators like `this`, `target`, and `args`. Pointcut designators like `within` and `withincode` do not lead to the generation of residues. They are statically evaluated (cf. above).

For all of these dynamic designators, residues are generated that rely on join point context access. Context access is achieved as described above using the `peek` instruction. An ensuing `instanceof` instruction performs the check, and conditional logic is used to skip the advice invocation, or to evaluate the next condition, if any.

The `endadvice` instruction marking the end of the advice block is used as the jump target for a failing test. It is guaranteed to lie behind the advice invocation. That way, the advice block can be generated without knowing the instruction it is going to be placed before.

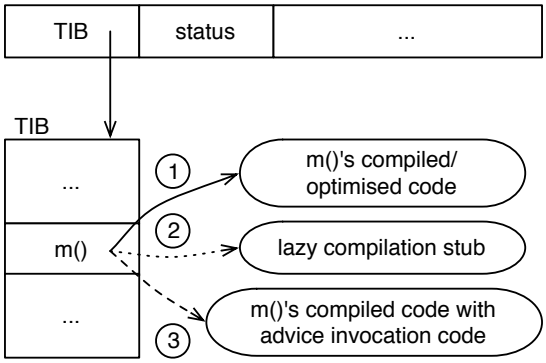

Figure 3.20.: Deployment of a class-wide aspect.

3.8.8. How Woven Code Takes Effect

Aspect weaving in Steamloom is done by first modifying affected methods' bytecodes using BAT and then scheduling them for lazy recompilation. An aspect's `deploy()` and `undeploy()` methods trigger all actions needed to activate or deactivate the aspect by iterating over all affected methods and appropriately changing their bytecodes.

To trigger recompilation the lazy compilation stub is reinstalled by having the corresponding JTOC and TIB entries point to it. Thus, baseline-compiled methods are invalidated and marked for recompilation to take place automatically the next time they are invoked, as with normal baseline compilation. This is illustrated in Fig. 3.20. At first (Fig. 3.20, index 1), the method pointer from the TIB points to the compiled code. Once the aspect is deployed, the method is invalidated and its code pointer now references the lazy compilation stub (index 2a). As soon as the method is invoked after that, the lazy weaving stub is executed (index 3).

Regardless of whether a method subject to weaving was compiled with the baseline or optimising compiler, it is always invalidated for lazy recompilation by the baseline compiler.

Concerning optimisation, Steamloom thus fully relies on the VM's AOS, which optimises methods based on profile data. An earlier version of Steamloom used immediate optimising recompilation for methods that were optimised prior to weaving. Since weaving—including the reinstallation of woven code—has to be atomic, this could lead to latencies because optimising recompilation is time-consuming.

Using the baseline compiler in any case is a better option, because it reduces weaving time to the time necessary to retrieve join point shadows and modifying method bytecodes. A method scheduled for lazy recompilation will only be recompiled when it is invoked for the next time. The AOS however knows about the method's state with regard to optimisation and exploits its knowledge to spawn an optimised compilation soon.

Special treatment is needed if a method that is to be decorated with advice code was inlined somewhere by the optimising compiler. In this case, it is not sufficient to simply invalidate the method since its native code may be inlined in various places all over the

```
REC = {m_0};
M = REC;
do
   M' = ∅;
   foreach m ∈ M do
      M' ∪= inline_locations(m);
   M = M' \ M;
   REC ∪= M;
until M = ∅;
```

Figure 3.21.: Algorithm to determine the set of invalidation candidates.

loaded classes. Instead, all inline locations of the method have to be invalidated as well, and all locations where those methods were inlined and so forth, resulting in a cascading invalidation.

A set of `VM_Method`s was added to each method, every element of which corresponds to an inline location of the method owning the set. Whenever an optimised method is invalidated due to weaving, an algorithm (cf. Fig. 3.21) determines the set of methods that also need to be invalidated due to inlining, and all such methods are invalidated and scheduled for lazy recompilation as well.

Starting from the method m_0 that is decorated with advice code the algorithm finds all methods that need to be invalidated and stores them in the set REC. The function *inline_locations* returns, for a given method m, all methods where m is directly inlined. The algorithm follows a generational approach, where m_0 forms generation 0, and all methods that directly inline a method from generation k belong to generation $k+1$. If a method m is inlined both in methods of generations a and b where $a < b$, m is defined to have generation b to avoid multiple inline location retrieval operations. Methods of generation k are stored in the set M. The inline locations of all methods in M are stored in M'. Next, all methods of generation $k+1$ are added to REC (these are the methods that are found in M' but do *not* belong to generation k). As soon as an empty generation $k+1$ is retrieved, the algorithm terminates.

Remarks on Weaving Semantics

With regard to dynamic weaving, two remarks are indicated that further describe the semantics of dynamic weaving in Steamloom. They address the range of methods that are affected by dynamic weaving, and the order in which advice are applied at join point shadows.

With respect to the effect of woven code, Steamloom adopts the following strategy. If an advice invocation is woven into the code of a method that is currently executing in some thread, then the advice will *not* take effect *until the method is executed the next time*. This approach was implemented to ensure the robustness of methods decorated by advice.

If, for example, an aspect attaches a before and an after advice to the execution of a

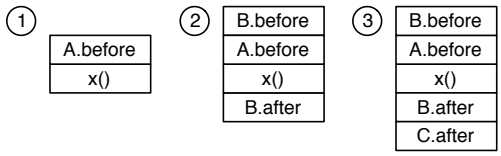

Figure 3.22.: An example for dynamic aspect precedence in Steamloom.

method which is currently active, then the after advice would be invoked, but the before advice would not. When the two advice logically belong together and even rely on each other, letting advice take effect *immediately* even in running methods would possibly leave the system in an inconsistent state.

A consequence of this approach is that a method cannot deploy an aspect that decorates join points which have shadows in that very method.

The advice application order at join points depends solely on the order in which the respective aspects are deployed. Steamloom at present provides no means to specify aspect precedence. Its default strategy is as follows: the advice pertaining to the aspect that is deployed *last* are the *outermost* with respect to application at join point shadows. The idea behind this is that a dynamically woven aspect always regards the join point shadow it decorates as a whole, including advice invocations that may already have been attached to it.

An example is visualised in Fig. 3.22. At index 1, a method call join point has already been advised with a before advice belonging to an aspect `A`. At index 2, another aspect `B` has been deployed that attaches a before and an after advice to the join point shadow. Its before advice *precedes* that of the aspect that was deployed first. Next, at index 3, an additional aspect `C` has been deployed that attaches an after advice to the shadow. This advice invocation is also added as a new outer "envelope" and is applied last.

3.9. Advice Instance Management

In this section, the concept and implementation of *advice instance tables (AITs)* will be introduced, which were developed and integrated with Steamloom. AITs provide an extremely efficient lookup mechanism for advice instances for both class-wide and instance-locally deployed aspects. Thread-local aspect deployment is not affected because it takes place at bytecode level, through conditionals.

AITs are an integral part of Steamloom's execution model and are not apparent at language level. This is the main reason for their superior speed: complex language-level data structures, being part of other dynamic AOP tools' infrastructures, are implemented at bytecode level, and thus access to them is subject to execution by the virtual machine. AITs, being part of the VM *itself*, do not suffer from that overhead.

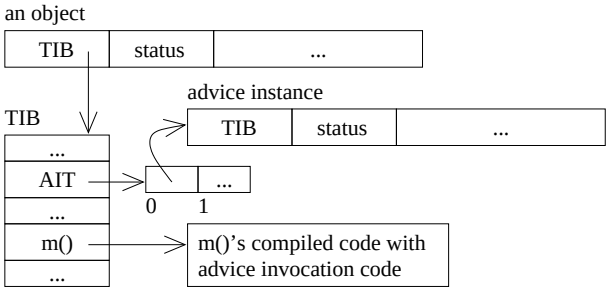

Figure 3.23.: Deployment of a class-wide aspect with AITs.

3.9.1. The Structure of Advice Instance Tables

Advice instances have to be associated with every class (for class-wide aspects) and with single instances (for instance-locally deployed aspects). Recall that, for every single class in the Jikes RVM, there exists a TIB. AITs are arrays of `Objects` to which references are stored in the TIB. This was achieved by extending the TIB data structure by one word containing the AIT reference. AIT creation takes place lazily, i. e., as soon as advice instances are registered for one particular class. Thus, most of the AIT references in the various TIBs are `null` most of the time. The elements contained in the AIT are references to the various advice instances that are relevant to the respective class in whose TIB the AIT is stored. The AIT is created with a default size and is dynamically expanded if necessary.

The overhead introduced by adding AITs to the virtual machine is low. Given that all instances of a class share the class's TIB, the overhead basically consists of one additional word per loaded class. The boot image size of the virtual machine is increased by 66 kB due to the addition of AITs, which is essentially also the increase of the memory footprint at startup.

Fig. 3.23 displays the situation *after* the deployment of a class-wide aspect. The aspect affects the implementation of the method `m()` of a given class. The AIT field in the class's TIB references an array of `Objects`, the first entry of which is the advice instance on which the actual advice method is to be invoked. The new code of the method `m()` now contains instructions that load the advice instance from offset 0 of the AIT and invoke the advice method thereon (for details on the new code see below).

If an aspect is deployed instance-locally, a deep copy of the class's AIT is made for the affected instance during the process of cloning the TIB (cf. Sec. 3.10). That way, advice instances belonging to class-wide aspects can still be looked up, while advice instances local to the object in question are not stored in the class-wide AIT.

Assume that one single instance named `o2` of the class known from Fig. 3.23 is now decorated with an additional aspect that affects the implementation of the method `n()` of that class—but only for the instance `o2`. The situation after deploying this instance-local aspect is shown in Fig. 3.24. The class's TIB was cloned and made `o2`'s local TIB. Furthermore, a deep copy of the class's AIT was created that is now referenced from `o2`'s TIB. Note that the version of the method `m()` referenced from the cloned TIB is that

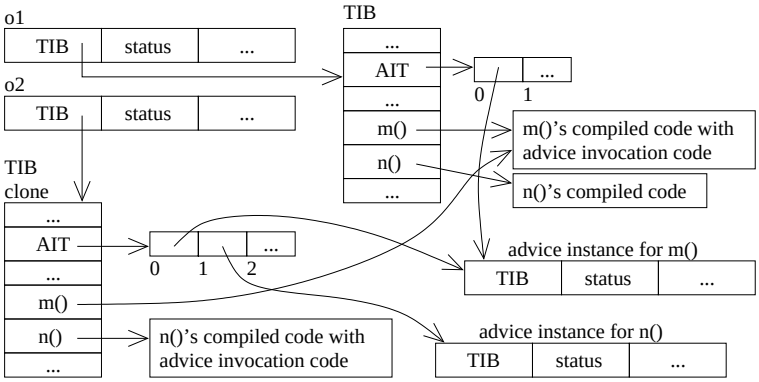

Figure 3.24.: Deployment of a class-wide aspect and an instance-local aspect with AITs.

of the class, and that the first entry in `o2`'s AIT refers the appropriate advice instance. Thus, the class-wide aspect is still active for the method `m()`. For the instance-locally decorated method `n()`, however, a new version was compiled and a new entry in the instance-local AIT was created.

3.9.2. Compiling Advice Instance Lookups

With AITs, weaving logic still operates at bytecode level, having to insert bytecodes to look up advice instances in the AIT. To implement such lookups, a new bytecode named `aaitpush` was added to Steamloom. This bytecode has one parameter, namely the AIT index from which an advice instance has to be loaded to be placed on top of the stack. After that, the object can be subject to method invocation in the usual way. The `aaitpush` bytecode is only used Steamloom-internally.

Of course, there has to be some kind of mapping from aspect units to AIT indices. This is achieved through a hash table that uses *aspect units* as keys. An aspect unit is a part of the Steamloom data structures, namely a container associating a given pointcut with an advice and, in case of instance- or thread-local deployment, with the respective instance or thread. The hash table maps aspect units to AIT offsets. The comparatively expensive hash table lookup is performed at JIT compile-time only. This is especially important for instance-local aspects, for which the old Steamloom version had to perform a hash table lookup every time the advice was to be executed.

Both the baseline and optimising compiler were modified to generate appropriate native or intermediate code from the `aaitpush` bytecode. The baseline compiler generates (for an x86 processor) exactly two machine instructions per `aaitpush`: one to look up the AIT reference in the TIB, and one to load the advice instance from the AIT. For the optimising compiler, two high-level intermediate instructions [36, 154] with the same semantics are generated. Because the `aaitpush` bytecode is directly transformed into high-level intermediate code, and no further modifications were made to the optimising compiler, all subsequent optimisations that the compiler performs—e. g., copy propagation or instruction reordering [154, 36]—are applied to the resulting code.

```
1 iload          <adviceId>
2 aload_0
3 invokestatic   <getAdviceInstance>
4 invokevirtual  <adviceMethod>
```

(a)

```
1 aaitpush       <index>
2 invokevirtual  <adviceMethod>
```

(b)

Figure 3.25.: Bytecodes for advice invocation; (a) without, (b) with AITs.

The brief bytecode snippets in Fig. 3.25 show the different approaches to weaving in Steamloom without (Fig. 3.25a) and with AITs (Fig. 3.25b). In both cases, the code shown is the code needed to invoke a `void` virtual advice method that takes no parameters. Steamloom without AITs had to call the static method `getAdviceInstance()`, which was part of the Steamloom infrastructure, to retrieve the appropriate instance for a given advice and, in case of instance-local deployment, an object. The method `getAdviceInstance()` retrieves the advice instance from the global array of advice instances. After that, the advice method can be invoked.

Using AITs, the woven bytecode sequence is significantly shorter, consisting of only two instructions. A considerable advantage of the dedicated `aaitpush` bytecode is that its translation is simple as it is a mere shortcut for some machine instructions that are (by the baseline compiler, for an x86 processor) generated as follows:

```
1 MOV  ECX, [<AIT reference address>]
2 PUSH [ECX + <index>]
```

Contrariwise, expressing the advice instance lookup in application-level byte codes results in the compilers generating various type checks, guards and array bounds checks that can be avoided for AIT lookups because the Steamloom environment guarantees type safety and array integrity.

The native code first retrieves the AIT reference from the TIB slot where it is stored. In the second step, an array lookup, taking the AIT reference as base address, is performed to retrieve the advice instance.

The address of the AIT reference slot is "hard-wired" into the generated code. This can safely be done because the Jikes RVM (and therefore Steamloom) stores TIBs in "immortal space" (cf. Sec. 3.3.7). Objects stored in immortal space are never moved around during garbage collection, so references to such objects can be hard-wired. Note that it is *not* the AIT which is placed in immortal space; it is the class' TIB. AITs are, from the memory management perspective, ordinary objects that live on the heap.

3.10. Scoped Aspects

Scoping is not to be confused with aspect instantiation granularity control. The latter, as found in, e. g., AspectJ, allows for associating dedicated advice instances with single objects, threads, or even control flows. In AspectJ, `perthis`, `pertarget`, `percflow`, etc. can be used to achieve this. Steamloom does *not* support this kind of control.

This section is about Steamloom's dedicated support for aspects that have a very narrow scope of applicability, e. g., a single *object* or *thread*. The former can be interesting in the context of role models where objects evolve with respect to the roles they play. The latter is of interest in Steamloom itself, as will become clear in Sec. 3.11: a `cflow` must always be matched in the context of the thread where it was started.

In the following, Steamloom's support for both kinds of scoping will be presented in detail.

3.10.1. Instance-Local Aspects

Contrary to AspectJ and others, Steamloom has no dedicated support for aspect instantiation control like with the AspectJ constructs `perthis`, `pertarget`, and so forth.

As mentioned above, associating advice instances with objects using `per...` constructs is *not* instance-local aspect deployment: advice functionality defined by *per*-instance aspects takes effect at all matching join point shadows, regardless of the currently executing object. The important feature of the mechanism is that all advice code is executed in a context that is specific to particular application objects—namely the aspect instance the respective advice method is invoked on. This allows for creating and controlling aspect state on a per-application-object basis. In contrast to that, in the context of an *instance-local* aspect, an application class's methods are decorated with advice functionality *only* if the methods are invoked on the object(s) affected by the aspect.

Scoping aspects to single instances is also not to be confused with *association aspects*, a concept that was implemented as an extension to AspectJ [102]. Association aspects are used to couple instances or groups of objects with others by means of an aspect that takes care of the association. In that, an association aspect has a somewhat richer semantics than a scoped aspect in the sense of Steamloom. The latter restricts the applicability of an aspect to a single objects, while the former coordinates groups of objects.

Instance-local aspects are dealt with in the following way: when an aspect is deployed that affects only a particular instance of a class, simply modifying the method is not feasible since it would affect the whole class. Instead, Steamloom exploits the fact that every object carries a TIB reference in its header. Usually only one TIB exists per class and is pointed to by all instances of that class. Furthermore, only one instance of `VM_NormalMethod` exists for every implemented method. Steamloom clones the affected object's TIB and lets the object reference the clone. The `VM_NormalMethod` object in question is also cloned and the clone's code is modified, so that two versions of the method in question exist: one for unaffected objects, and one for those objects that are in the scope of the deployed aspect. Recompilation takes place as described above.

In Fig. 3.26, instance-local aspect deployment is illustrated. Initially, the TIB pointers of both objects `o1` and `o2`, instances of the same class, reference the same TIB (index 1). Upon deployment of the aspect on `o2`, both the TIB and the decorated method are cloned and the TIB's respective entry is changed to point to the lazy compilation stub (index 2). When the method is next called, it will be compiled and the instance-local TIB entry will point to the native code (index 3).

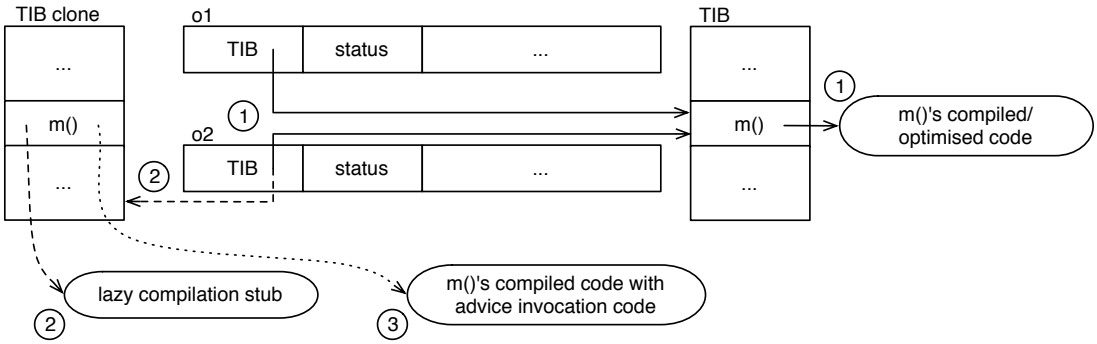

Figure 3.26.: Deployment of an instance-local aspect.

If a method decorated by an instance-local aspect was inlined in some place, it is not any longer looked up via the TIB: its native code is directly executed in place. In the context of instance-local aspects, at least two versions of a given method exist—the original one and the one that was cloned for the affected instance. Since the optimising JIT compiler cannot decide which version of the method in question is to be inlined (the original, or the instance-local one), inlining is prohibited for such methods [32].

The case that a method is advised by both class-wide and instance-local aspects is not fully covered in the current implementation. While it is possible to add instance-local advice to a method that is already subject to class-wide advice, support for arbitrarily adding and removing all kinds of (instance-local or class-wide) advice is not implemented. Since AIT slots are recycled to avoid unnecessary array growth, mixing class-wide and instance-local advice instance references in one single array per class leads to unclear slot allocations. The only kinds of join point supported with instance-local decoration are method call and execution join points.

3.10.2. Thread-Local Aspects

When an aspect is deployed thread-locally, this means that its advice only take effect if the join points they are attached to occur in that very thread, while the aspect does not have an effect on the application in all other threads. This is not only useful in contexts where functionality running in a single thread has to fulfil requirements that are implemented as crosscutting concerns. It is also of interest in the context of `cflow` management (details on this are given in Sec. 3.11). In addition, thread-local deployment is a language feature in some cases (e. g., in CaesarJ, cf. Sec. 2.3) that deserves implementation effort.

Aspect applicability to threads is checked through residues. In pseudo-Java syntax, a residue for thread applicability of an advice invocation looks as follows:

```
1 if(VM_Thread.aspectUnitEnabled(<aspect unit ID>))
2   <invoke advice>;
```

That is, the residue asks the VM itself for applicability; no application-level entities are involved in this check.

The VM's internal representation of threads, the `VM_Thread` class, was extended to support thread-locality residues. The static method `aspectUnitEnabled()` accepts an `int` value, namely the ID of the aspect unit whose advice invocation is to be checked for applicability. This ID is known at compile-time and hard-wired into the woven code.

Each `VM_Thread` instance, i. e., each thread, moreover maintains a `boolean` array mapping aspect unit IDs to Boolean values denoting whether a particular aspect unit is applicable in the thread. The array is updated during deployment and undeployment. The array size is initialised with a fixed value, and the array grows dynamically as it is needed. Since aspect unit IDs are recycled, so are the array slots.

Thread locality information is stored in the internal representation of threads, but it is made accessible to residual code through the internal representation of the virtual processor, `VM_Processor`. A field was added to that class that references the aspect unit ID `boolean` array of the thread currently running. The VM's scheduler was modified to update that reference upon every thread switch.

It would have been possible to store the instance in `VM_Thread` only, but accessing a field of `VM_Processor` requires one less instruction to load the value. The current processor can be obtained by invoking `VM_Processor.getCurrentProcessor()`, while the current thread can only be retrieved from the current processor object.

When the static method `aspectUnitEnabled()` is invoked from a thread-locality residue, it retrieves the `boolean` array from the `VM_Processor` and returns whether the passed aspect unit ID is applicable. Depending on the return value, the advice is invoked or its invocation is skipped.

3.11. Support for `cflow`

The implementation efforts dealing with dynamic pointcuts in Steamloom have been focused on support for `cflow`. All in all, three experimental implementations have been provided that will be presented in this section. An evaluation of their performance will be given in Sec. 4.2.6.

When `cflow` is used, the idiom `cflow(pc1) && pc2` is frequently met, denoting that the pointcut shall match join points pertaining to `pc2` *only* if they occur in the control flow of some join point matched by `pc1`. In the following, join points matched by `pc1` will be called *control flow constituents*. A control flow constituent's shadows mark *entries* and *exits* of control flows. Shadows pertaining to join points matched by `pc2` will be called *dependent* shadows.

An implementation of `cflow` needs to address the following two issues:

1. At control flow entries and exits, action needs to be taken to monitor the state of the control flow, i. e., whether it is active or not.

2. At dependent shadows, it must be checked whether the control flow is currently active to determine whether the advice attached to the join point shadow needs to be invoked.

Three different approaches for both of these points have already been met in the presentations of AOP implementations in Ch. 2. They shall now be outlined in generalised form before their implementation in Steamloom is introduced.

3.11.1. Approaches to Implementing `cflow`

Counters

When this approach is adopted, residues are attached to control flow entries and exits that update counters. When a control flow is entered, the counter is incremented, and it is decremented when the control flow is left. At dependent shadows, residues are woven that check whether the counter is greater than zero. If so, the control flow is active and the advice can be executed.

It is not sufficient to employ a simple `boolean` to note whether a given control flow is currently active. Using a `boolean` value, it is not possible to determine whether a control flow has been entered recursively.

Control flow counters exist once per control flow. Furthermore, they must be thread-local for this approach to work. Were they not, different threads entering and leaving the same control flow could easily corrupt the control flow counter state. For example, AspectJ (cf. 2.2) uses `ThreadLocal` instance to maintain control flow counters.

Using counters imposes a constant overhead at control flow entries and exits as well as at dependent shadows.

This approach is, of the systems presented in Ch. 2, also used by CaesarJ, AspectWerkz, and Reflex.

Stack Walking

The stack walking approach does not need any residues at control flow entries and exits. Instead, it gets hold of the current call stack at dependent shadows and iterates over the methods on the stack to check whether the control flow in question is currently active.

This approach does not need to regard thread locality, because the call stack that a residue accesses is *always* the one of the currently executing thread. Depending on the language used, there are different approaches to access the call stack. In Java, the call stack can be accessed by creating an instance of `Throwable`, which can be queried for the stack frames via its `getStackTrace()` method. In Smalltalk, the call stack is immediately accessible due to the reflective nature of the language. Any approach based on C can operate on the machine level directly, which however reduces portability because stack frame layouts may differ depending on the actual environment.

There is no cost at control flow entries and exits connected with stack walking. However, the cost imposed on dependent shadows is directly dependent on the depth of the call stack. In the most inauspicious case, the entire stack must be parsed only to determine that a particular control flow is *not* active at present.

On the other hand, stack walking could be beneficial when complex nested control flows are to be matched, stating, for example, that a given sequence of methods must

be on the stack in a given order. It is possible to regard such a nested control flow as a regular expression that can be matched by an automaton that walks the stack.

The systems that use stack walking are Arachne, JAsCo, PROSE, Spring AOP, and AspectS.

Continuous Weaving

Using continuous weaving, it is possible to leave dependent join point shadows *completely* unaffected while the control flow in which they should be decorated with advice is inactive. Instead, the control flow entries and exits are decorated with residues that trigger continuous weaving of advice invocations at dependent shadows. In this case, there is still an element of residual logic required at dependent shadows: the advice must only be invoked when the shadow is reached in the same thread in which the control flow is currently active.

The simplest approach to implementing `cflow` using continuous weaving is to decorate, when the control flow is entered *all* dependent shadows at once, and to withdraw the woven code when the control flow is left. In the spirit of continuous weaving, more fine-grained approaches are imaginable.

The cost imposed on control flow entries and exits, or on parts of the dependent shadows, is as high as that of dynamic weaving. At dependent shadows, an additional overhead is introduced to check for thread applicability.

None of the systems presented in Ch. 2 employ continuous weaving. AspectS has, in principle, support for doing so, though.

3.11.2. Support for `cflow` in Steamloom

In Steamloom, all of the three aforementioned approaches have been implemented. All of the implementations exploit the fact that Steamloom is a VM extension in that they rely on specific functionality offered by the virtual machine, or in that they themselves integrate part of their functionality in the VM. The three implementations will be presented in the same order as above.

Counters

Control flow counters are, in Steamloom, managed in `CflowCounter` instances. The `CflowCounter` class is trivial: it has a single public member of type `int` that carries the counter value.

In Lst. 3.13, pseudo bytecode is shown that illustrates the shape of woven code at control flow entries and exits. A before and an after advice are woven. The before advice block pushes, like a normal advice block, an advice instance on the stack. The advice instance is an object of the type `CflowTracker`, on which the `enterCflow()` method is invoked. The value returned from the method is the `CflowCounter` instance responsible for this control flow. It is stored in a local variable. The after advice block invokes the `exitControlFlow()` method on the control flow tracker and passes it the counter object, which it restores from the local variable.

```
1  ...
2  beginadvice          <id>
3  aaitpush             <AIT handle>
4  invokevirtualadvice <CFlowTracker.enterControlFlow()>
5  astore               <local>
6  endadvice            <id>
7  ...                                ;; instruction(s) constituting the control flow
8  beginadvice          <id>
9  aaitpush             <AIT handle>
10 aload                <local>
11 invokevirtualadvice <CFlowTracker.exitControlFlow()>
12 endadvice            <id>
13 ...
```

Listing 3.13: Code woven at control flow entries and exits in Steamloom (counter approach).

Instances of `CflowTracker` are created during weaving, when a `DeployedAspectUnit` is processed that contains a `cflow` designator (cf. Sec. 3.8.1). They encapsulate a number of `CflowHelper`s that themselves know to which dependent shadows they are attached.

In `enterControlFlow()`, the control flow tracker iterates over all of its helpers and asks them to update the aspect units associated with them (see next paragraph). The control flow counter is also incremented. Correspondingly, `exitControlFlow()` decrements the counter and also notifies the helpers.

The aspect units that have to be updated are instances of `DeployedAspectUnit` logically representing decorated join point shadows. Updating means that they are registered or unregistered with the thread in which the control flow is entered or left.

The aspect units related to `cflow` are treated in exactly the same way as those used with thread-local deployment (cf. Sec. 3.10.2): when the control flow they depend on is entered or left, the corresponding `VM_Thread`'s `boolean` array storing the applicability of aspect units is updated accordingly.

This means of course that the same residue that is used for thread-locally deployed aspects can be used for `cflow` checks. Thus, the code woven at dependent shadows when the counters approach to implementing `cflow` is used has the form shown in Lst. 3.14. This is the exact counterpart of the Java snippet shown in Sec. 3.10.2.

```
1  ...
2  beginadvice     <id>
3  iconst          <aspect unit ID>
4  invokestatic    <VM_Thread.aspectUnitEnabled()>
5  ifeq            ;; jump to endadvice instruction if test fails
6  invoke...advice <advice method>
7  endadvice       <id>
8  ...
```

Listing 3.14: Code woven at control flow-dependent shadows in Steamloom (counter approach).

Stack Walking

As mentioned above, no residues are required at control flow entries and exits when stack walking is used to implement `cflow` checks. Consequently, Steamloom only weaves code at dependent join point shadows that has the form shown in Lst. 3.15. The code differs from the code woven with the counters approach only in the method that is invoked to perform the check.

```
1 ...
2 beginadvice     <id>
3 iconst          <aspect unit ID>
4 invokestatic    <StackFrameMatcher.isInControlFlow()>
5 ifeq            ;; jump to endadvice instruction if test fails
6 invoke...advice <advice method>
7 endadvice       <id>
8 ...
```

Listing 3.15: Code woven at dependent join point shadows in Steamloom (stack walking approach).

A `StackFrameMatcher` is created for a `cflow` designator when the latter is deployed. The matcher is initialised with the pointcut designator denoting the constituent join points and the advice attached to the dependent shadows. From this information, it builds, internally, a `StackPattern` instance that represents the stack layout (in terms of methods on the call stack) that must be met in order for the constituent pointcut to match. In case of nested control flows, the pattern contains the methods constituting the nested control flow in the given order. Each entry of the pattern can—if the corresponding constituent pointcut contains wildcards—match multiple methods.

The `isInControlFlow()` method accepts, like the other methods responsible for control flow checking, the ID of the aspect unit whose advice is attached to the join point shadow in question. It retrieves the stack frame matcher responsible for that shadow from an internal hash map and asks it to walk the stack and match it against the pattern it contains.

The matching process extracts the IDs of compiled methods from the VM-internal stack frames. From the ID of a compiled method, the internal representation of the method, in the form of a `VM_Method` instance, can be resolved. The methods retrieved from the stack frames are subsequently matched against the elements of the stack pattern to check. As soon as the pattern is safely identified, the process stops, and the advice can be invoked.

Continuous Weaving

Continuous weaving as implemented in Steamloom follows the aforementioned simple approach where all dependent shadows are immediately decorated when the control flow is entered.

When an aspect unit containing a `cflow` pointcut is deployed, the entry and exit shadows are decorated with code that looks exactly like the code shown for the counters approach in Lst. 3.13. Most of the dependent shadows are, however, *not* decorated. Some are, which is due to a small optimisation that will be explained below.

The difference lies in that the `enter...` and `exitControlFlow()` methods, apart from updating counters, trigger the weaving of the associated aspect units that pertain to dependent shadows. Counters still need to be managed because a control flow may be entered recursively; the dependent shadows must be kept decorated until the last activation of the respective control flow is left.

Dependent shadows are completely unaffected by residues pertaining to a `cflow` pointcut until the first thread enters the control flow, which triggers the deployment of residues and advice invocations at dependent shadows. All other threads that enter the control flow merely lead to the respective aspect units being marked as applicable in their respective `boolean` arrays. Consequently, code woven at the dependent shadows is not removed as long as *any* thread is inside the control flow. When the last thread leaves the control flow, residues and advice invocations at dependent shadows are undeployed.

The code woven at dependent shadows also has exactly the shape that it has when the counter approach is used. This is needed because thread-locality must be ensured: all control flows are, as mentioned above, inherently thread-locally decorated.

The aforementioned optimisation applied in the implementation of continuous `cflow` weaving is applied when a situation like the following one is met. When a pointcut like

```
1 cflow(execution(void X.m())) && call(void Y.n())
```

is used *and* the method `X.m()` contains calls of `Y.n()`, then the method whose execution constitutes the control flow also contains dependent shadows.

Normally, weaving would add a before and after advice to the execution of `X.m()` that dynamically deploys the residues and advice invocations at the dependent shadows. However, since the method `X.m()` itself contains such shadows, they can right away be decorated as well. This is just what the optimisation is about.

It applies a very simple form of static analysis and determines exactly those dependent shadows that are contained in methods constituting the control flow. That way, an unnecessary weaving step can be avoided for dependent shadows that are guaranteed to be reached.

Limitations

Steamloom's support for `cflow` is currently limited. It can match control flows, but it does not allow for accessing state from the constituent join point shadows. Hence, constructs like the following are not possible:

```
1 call(void X.x()) && cflow(execution(void Y.y()) && target(y))
```

In this AspectJ example, an advice would have access to the instance of `Y` on which the execution of `y()` constitutes the control flow.

This kind of context access requires the maintenance of stacks to monitor the state of control flows, instead of only monitoring their being entered and left. Control flow stacks have not been implemented in Steamloom.

4. Evaluation

4.1. Introduction

This[1] chapter is dedicated to the analysis and discussion of the AOP approaches presented in the previous chapters—related work from Ch. 2 and Steamloom. The evaluation aims at underpinning the claims made about Steamloom, namely that it supports dynamic AOP flexibly and efficiently, and at pointing out the cases in which it does so especially well, or less well.

The questions that are of interest when evaluating an AOP language implementation are manifold, covering all kinds of concerns ranging from implementation to usability issues. In this introduction to the evaluation, the criteria applied in the discussion will be described briefly. There are two levels of criteria that are applied at different degrees of abstraction. They are described as follows:

1. Technical criteria address implementation specifics in the discussed systems. They cannot be expressed in raw performance data but need to be addressed in an argumentative way instead. However, they still allow for classifying systems and grouping them into categories.

2. Performance criteria are expressible in raw numbers, i. e., in measurement results such as those from performance benchmarking experiments.

In the following, the criteria at these two levels are briefly described. For all criteria, the possible impact of characteristics at their particular levels on criteria at other levels is also briefly outlined.

It should be noted that all of the following criteria are, in the context of this work, looked at from a language *implementation* point of view, instead of a language *design* perspective. Thus, the usability of *languages* and their expressiveness are explicitly not in the focus of the discussion.

4.1.1. Technical Criteria

The criteria at this level are, unlike most of the performance criteria, not necessarily expressible in "raw numbers". They instead deal with implementation details of the various systems in question. Hence, they are still at a rather technical level, but are expressed in implementation details instead of measurement results.

[1] Part of the material in this chapter has previously been published [77].

Representational Overhead for Application Model Exposure Normally, the application model is used for exposing join points, evaluating pointcuts, and weaving. Such exposure may, depending on the way the AOP system is implemented, require dedicated data structures that in turn require additional memory and affect performance because they have to be maintained.

An AOP implementation should avoid too complex data structures for application model exposure as far as possible. If such data structures are inevitable, their maintenance should be very efficient.

Amount of Infrastructure Involved in Residue Evaluation and Advice Invocation This reflects on the amount of infrastructural code that has to be executed as part of the application. In the context of this work, the term "infrastructural code" describes all code pertaining to an AOP environment that is executed by the base language environment. The two typical uses of infrastructural code are residual logic and advice dispatch.

Obviously, the execution of infrastructural code may lead to a performance overhead, depending on its shape and size. It also affects debugging: for example, consider the case when a programmer wants to trace back an error occurring in an advice to the join point shadow in the base application where the advice invocation originated. It is not the programmer's intention to browse through a possibly deep portion of the call stack, consisting of invocations of infrastructural code belonging to the AOP implementation. Instead, the debugger should *immediately* point the programmer to the correct location.

Security This criterion addresses the security model of the Java programming language. The discussion is not about high-level security APIs, but about core features of low-level security on the Java platform. In particular, Java language security is comprised of the language's open definition, its static type system, automatic memory management and lack of pointer arithmetic, and bytecode verification. In the discussion, it is analysed in how far AOP implementations may affect it.

4.1.2. Performance Criteria

The criteria presented here address properties of AOP implementations that can be expressed in raw numbers, i. e., performance data. Performance is complex, though, and needs itself to be regarded at different levels of abstraction.

First of all, an AOP implementation augmenting a base language implementation may have a certain performance impact on a running application even if no aspects are used. This is called the *basic overhead* of the AOP implementation and has to be measured. Ideally, there should be no such overhead to avoid unnecessary performance penalties.

Apart from the overhead an AOP environment imposes on a running application *not* using aspects, the performance of applications *using* aspects is certainly also important. In this regard, there are two major points of interest, namely the cost of woven code on the one hand, and the cost of weaving on the other.

The cost of woven code is effected by all the elements that are woven into base application code at join point shadows, regardless of whether advice invocations are attached to a shadow or not. In this respect, it is interesting to know how expensive it is to have join point shadows decorated with woven code, including advice invocations. The cost of woven code impacts the overall performance of the running application.

The cost of weaving comprises of pointcut evaluation and the actions taken to let aspects take effect at join point shadows. These operations possibly have significant impacts on class loading and run-time performance, depending on the time at which the mechanisms are applied.

Another important concern is the amount of memory that is consumed when an AOP system is used to run an application with aspects. Since a high memory overhead may lead to more frequent runs of the garbage collector in Java environments, which may lead to application performance degradations, memory consumption has to be addressed.

Finally, it is interesting to know to what degree the employment of a particular AOP implementation influences the performance of a running application. This facet of performance is mostly expressed in responsiveness of running applications.

Such large-scale performance is influenced by the small-scale performance (i. e., that of core mechanisms described above) only to the degree that software actually relies on AOP mechanisms. When an application makes heavy use of aspects and a large number of advice is attached to many join point shadows, the impact will be visible. Where few interactions between the base application and aspects occur, the cost of invoking advice will certainly not be the bottleneck. Nevertheless, the performance of advice execution—with all its facets as suggested above—has an impact on the performance of applications.

4.1.3. Organisation

A bottom-up approach is followed to formulate an evaluation, giving this chapter the following structure. The performance criteria are addressed in detail in the very next section. After that, the technical criteria are discussed.

The discussions deal with the various criteria and the ways the regarded AOP implementations fulfil them. Each of the criteria will first be detailed in depth, followed by a description of how it is going to be assessed. Since different sets of AOP implementations are regarded in the different following sections, each of them will also give an overview of them. After that, assessment results are discussed for all of the systems that have been regarded. At the very end of this chapter, a summary will discuss evaluation results with respect to the level of integration with the underlying execution environment that the evaluated systems exhibit.

4.2. Performance Criteria Assessment

Performance measurements deal with various topics about the performance of AOP environments. In this introductory section, only an overview of the measurements that were made is given. The following sections will be more detailed: each of the following

questions is investigated in detail in a structured manner. A detailed description of the environment and configuration that was used to perform the particular measurement is followed by a presentation and discussion of the measurement results.

What is the common overhead of using an AOP environment? Even if an application is built using aspect-oriented techniques, considerable parts thereof may still not be affected by the applied aspects. It is of great interest that the performance of applications (or application parts) that are not subject to decoration with crosscutting functionality not be negatively influenced by the mere employment of an AOP environment. This especially holds in the case of a virtual machine, where there is no choice of switching AOP functionality on or off. This question is addressed by measuring the performance of applications running on "normal" virtual machines versus various environments with AOP extensions.

What additional cost is imposed on class loading? Some of the systems in question apply modifications to class and method bytecodes or build meta-information data structures as classes are loaded into the VM. In close conjunction with the previous question, the issue of class loading cost is investigated.

What is the cost of invoking advice? The ways advice invocations are implemented in the various AOP implementations as presented in Ch. 2 differ significantly. Therefore, the question of how the different implementation approaches affect performance is very interesting. This of course includes multiple kinds of join point shadows and multiple kinds of advice. It is also interesting to know how the way context extraction is performed to pass parameters to advice further affects performance.

What is the cost of deactivated aspects? Some AOP approaches that weave at load-time weave code into the base application that prepares join point shadows for the later attachment of advice. This code is called the "footprint" of deactivated aspects. It is always executed, even when there advice are never attached to the corresponding join point shadows. The footprint can have a significant impact on performance, depending on the nature of the code that is used.

What is the cost of applying scoping to aspects? The above question needs to be answered for specially-scoped aspects as well, i. e., for instance- and thread-local aspects. For these two, Steamloom has dedicated support built right into the virtual machine itself. Hence, the answer to this question gives an important contribution to the evaluation of VM-integrated support for AOP.

What is the cost of support for dynamic pointcuts? This question mainly addresses the cost of executing residual code of all kinds. The focus in discussing this question is on the support for the `cflow` pointcut, for which Steamloom has extensive dynamic support.

What is the cost of dynamic weaving? Those systems that allow for dynamic weaving have to be closely examined with regard to this question. It is important that the stall imposed on the application during weaving be as short as possible. The cost of dynamic weaving comprises the cost of actually finding the appropriate join point shadows in code, that of modifying method bytecodes, and that of letting the modifications take effect.

What impact has the use of aspects on memory consumption? An AOP system normally needs some memory to store internal information. The impact on the application should be as low as possible to avoid performance penalties due to frequent garbage collection runs.

How efficiently is an aspect-oriented application run? The introduction to this chapter mentions that large-scale performance measurements are needed apart from regarding single mechanisms such as weaving and advice invocations. Indeed, it is interesting to know how well a complex aspect-oriented application performs when it is run on several AOP implementations. Such an application should make extensive use of "production aspects" to realise some of its crosscutting concerns.

Unfortunately, there is no actual benchmark application or suite that covers large-scale performance in conjunction with AOP, and that is available on a broad basis, i. e., provides the benchmarks for a wide range of AOP implementations. Such a benchmark would have to cover applications that make *actual use* of aspects implementing crosscutting concerns that are *real parts* of the applications, instead of aspects that are merely added as a kind of "patches" to see, e. g., how many method invocations there are in the application. In a nutshell, the aspects employed in such a benchmark would have to be complex on the one hand, and a real contribution to the application itself on the other.

First steps towards a large-scale AOP benchmark have been made by Dufour et al. in conducting an extensive analysis of the dynamic behaviour of AspectJ programs [54]. While the analysis is mostly focused on profiling woven code to gain the amount of instructions that are executed for various AOP-induced reasons, the applications can well be used as performance benchmarks.

Since AspectJ was in the focus of their work, Dufour et al. have certainly used AspectJ's language features to a great extent, which makes porting all of the benchmarks to all other AOP implementations in question impossible. For example, systems like PROSE and Steamloom do not at all support static crosscutting. Most of the benchmarks in Dufour et al.'s work use introductions, `declare parents` statements and so forth. Hence, these are not usable to compare the performance of Steamloom to that of other AOP implementations.

Due to these problems, large-scale performance measurements are not covered in this work. They remain an important issue for future work, though, directions for which will be briefly discussed in Ch. 5.

4.2.1. Measurement Environment

For better comparability, the measurements were restricted to Java-based systems. Hence, AspectS and Arachne are not regarded at all below. The following versions of the various systems have been used for the measurements:

- AspectJ 5 (milestone 4), and AspectJ 1.2,
- AspectWerkz 2.0,
- CaesarJ 0.7.0,
- JAsCo 0.8.6,
- PROSE 1.3.0,
- Reflex 3.0-alpha31, and
- Spring AOP as contained in version 1.4.2 of the Spring framework.

All performance measurements were run on a Dual Pentium IV Xeon (3 GHz each) workstation with 2 GB RAM running Linux 2.4.23. The heap size of all involved JVMs was set to an initial 512 MB and to adaptive resizing. Where other settings were used, this will be mentioned in the appropriate place below.

4.2.2. Overhead Measurements

With regard to the basic overhead of AOP systems, two concerns are regarded: running complex applications and class loading. As mentioned in the introduction to this chapter, it is of general interest to know to what extent employing a run-time environment with AOP functionality influences the general performance of running applications that do *not* exploit the AOP facilities. In this section, the results of measurements performed for the various systems in scope will be discussed. The interest in class loading performance stems from the fact that some AOP systems employ modified class loaders, which may impact the start-up performance of an application.

The systems taken into account in the overhead measurements are those that either attach some additional AOP-related functionality to the class loading process, or that consist of modified run-time environments. Hence, AspectJ, CaesarJ, and Spring AOP are not represented here.

Any overhead is a relative value with respect to some reference. For AspectWerkz, JAsCo, PROSE and Reflex, the reference for computing overheads is the Java 5 standard VM. For PROSE/Jikes, the Jikes RVM 2.3.0.1 was the reference, and for Steamloom, it was Jikes 2.3.1.

Benchmarks for Overhead Measurements

For evaluating the performance of Java execution environments not employing AOP, there a wealth of benchmark suites is available. Two major standardised suites from the SPEC group were chosen for this work: the SPECjvm98 [139] and the SPECjbb2000 [138] benchmarks.

SPECjvm98 The SPECjvm98 benchmark suite consists of several applications that impose different stresses on the JVM running them. Some of the applications are "number crunchers", e. g., the "compress" benchmark that performs data compression using the Lempel-Ziv algorithm, or "mpegaudio", which decodes a MP3 file to raw audio data. Others create a high memory load, like "db", which creates an in-memory database and executes several queries, and "jess", which executes an expert system to solve logic puzzles. In "mtrt", a ray tracer is run, and "jack" and "javac" generate parsers and compile Java source code. More details on the natures of the several benchmark applications are available on the SPECjvm98 benchmark home page[139].

For each of the benchmarks, three different problem sizes exist: 1, 10, and 100. The sizes 1 and 10 exist for testing purposes, size 100 is intended for the generation of actual benchmark results. The SPECjvm98 results described in the following were obtained by running each of the benchmark applications 20 times at problem size 100.

SPECjbb2000 Other than SPECjvm98, the SPECjbb2000 benchmark consists of a single application that is albeit very complex. The benchmark emulates a 3-tier system, in which business logic and object manipulation create the greatest workload. The clients accessing the server do not exist as physical entities; they are simulated by driver threads. Also, there is no actual database; it is simulated by binary trees.

The benchmark creates an increasing number of so-called "warehouses", i. e., databases. For each number of warehouses, a two-minute measurement is executed where a client thread per warehouse accesses the latter's data. The benchmark yields a "score" that expresses the throughput in SPECjbb2000 operations per second.

For the measurements with SPECjbb2000 in the context of this work, a configuration with a maximum of 8 warehouses was used. The benchmark was, on all systems, run with two memory configurations, namely with the heap fixed to 512 MB and 1 GB, respectively.

Results The results for *SPECjvm98* are shown in Fig. 4.1. The figure does not show results for particular benchmark applications, but average results gathered by running all benchmarks on the systems listed in the figure.

Two values are shown for each system: the overhead exhibited by the system during the *first* of the 20 runs of the benchmarks, and the average overhead, computed over all 20 runs. The first run is interesting because class loading only occurs during this run; observing its performance in particular gives an impression of the impact of a modified class loading process on overall performance.

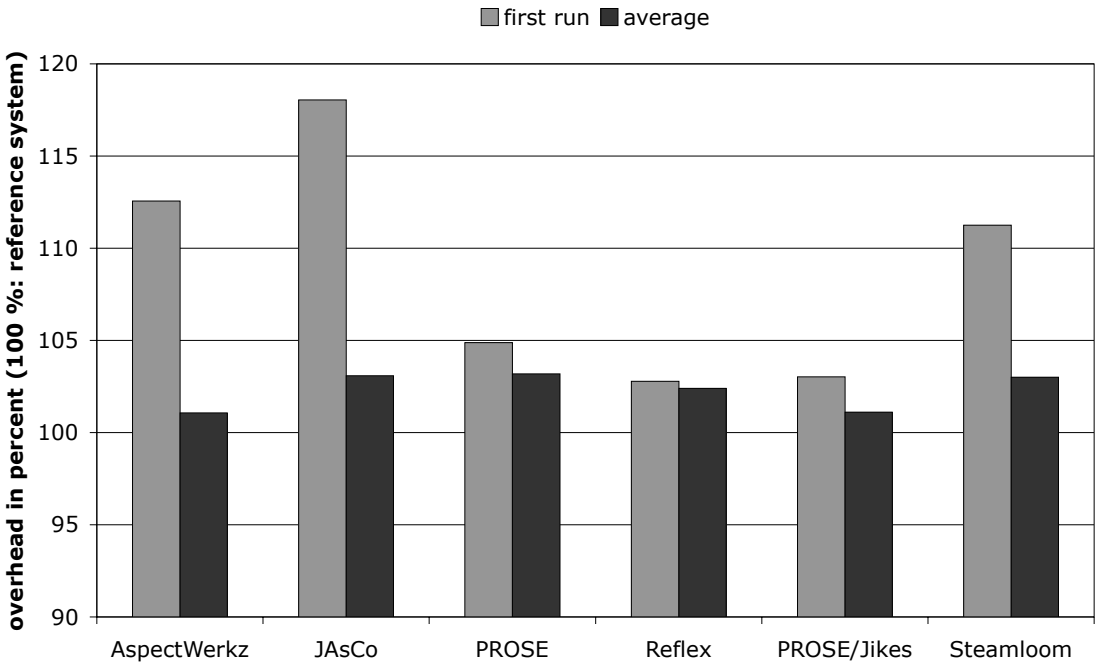

Figure 4.1.: Results from SPECjvm98 overhead measurements.

The first run of the benchmark however still also reflects on the performance of the VM during the run, which is why measurements concerning class loading have been conducted. They will be presented in a dedicated section below.

For all systems, the overhead is larger for the first run than it is on average. This was to be expected. On the one hand, VM optimisations cannot take as much effect at this time as they can in later runs. On the other hand, class loading occurs during the first run only.

All systems in question have some form of addition to the class loading process, even those that, like PROSE and PROSE/Jikes, do not employ a modified class loader. Still, even these systems have to check whether already deployed aspects apply to dynamically loaded classes.

The overhead in the first run is largest in AspectWerkz and JAsCo. Both of them significantly augment class loading. In AspectWerkz, *every* class is transformed to its ASM representation, and transformation context objects are created for it, regardless of whether there are any aspects or deployment scopes registered at all. In JAsCo, a large amount of checks is performed, also regardless of the presence of aspects.

Reflex is an exception to this observation. When there are no configurations of any meta-object protocols given, its class loading process is extremely efficient. The Reflex class loader is a specialised Javassist class loader. In a Javassist class loader, so-called *translators* can be registered that describe how classes are to be modified during the loading process. The Reflex harness installs its own translator in the Javassist class loader. All reflective links to be installed during class loading are registered with it. During

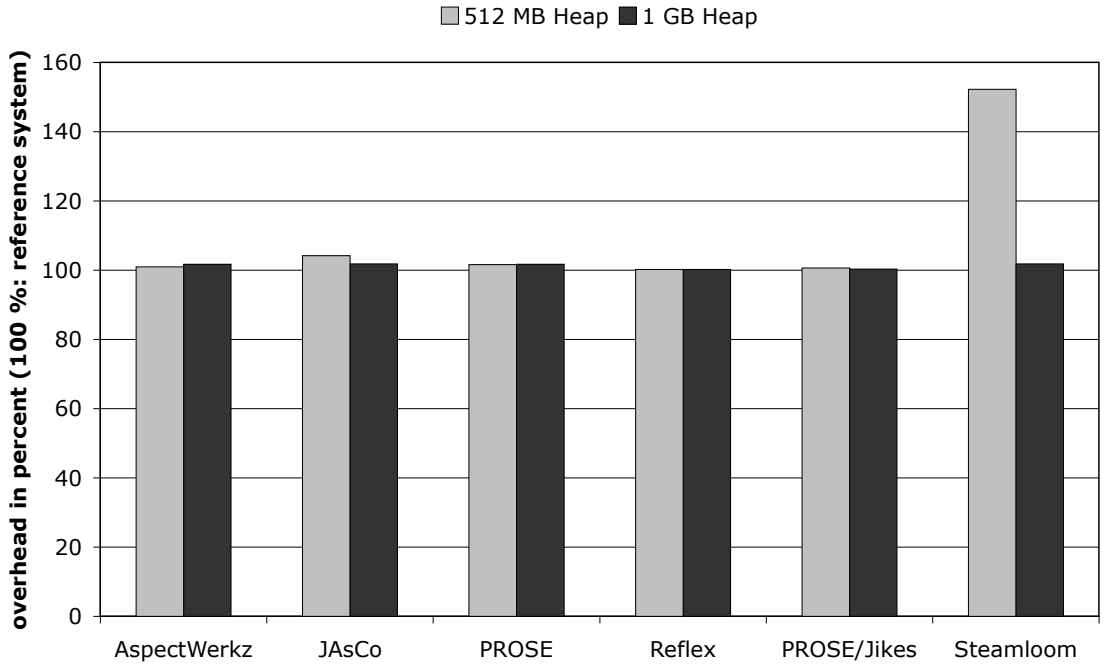

Figure 4.2.: SPECjbb2000 overhead measurement results.

class loading, it simply checks whether *any* links are installed—if not, it immediately delegates class loading to the parent class loader.

In the case of Steamloom, the overhead during the first run is due to the construction of linked lists of instruction objects from class files, and from the index building process for join point shadow retrieval, which is performed during class loading. JIT compilation, slightly more expensive due to the new bytecode stream implementation, also carries weight.

As for the average performance, all systems but AspectWerkz exhibit a certain overhead. The overheads are reasonably small in all cases, the highest observed overheads amounting to about 3 % for PROSE, JAsCo and Steamloom.

The *SPECjbb2000* results are displayed in Fig. 4.2. All systems exhibit very low overheads for both heap sizes, if any. Steamloom is a notable exception. For the 512 MB heap, it exhibits an overhead of 52 %. This is due to Steamloom creating a large amount of objects to represent method instructions (*each* bytecode instruction is represented by one object). Since, in the Jikes RVM, VM-internal and application data structures share the same heap, the garbage collector is responsible for both. SPECjbb2000 causes extreme memory loads, which is why the heap is frequently exhausted in Steamloom, where it contains a large number of instruction objects. With a 1 GB heap, Steamloom also performs well because the heap is less frequently exhausted.

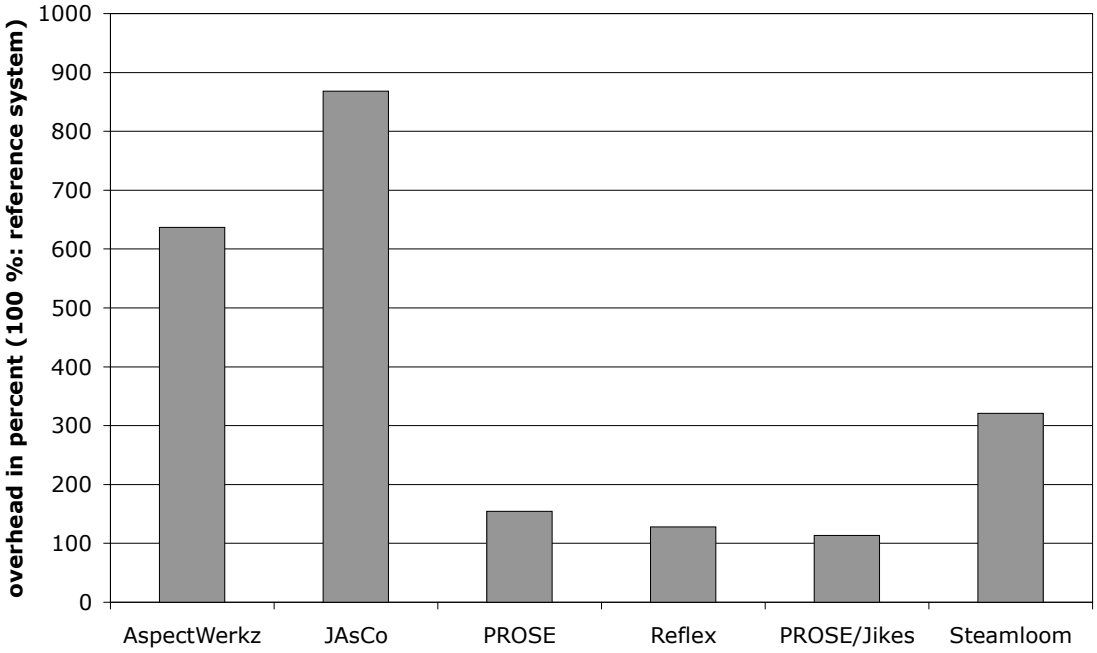

Figure 4.3.: Results from class loading overhead measurements.

Class Loading

To measure class loading performance in isolation, a simple application was used that loads a selection of classes from the SPECjvm98 benchmarks by explicitly loading each of them via `Class.forName()`. The time used for this was measured.

The classes selected for the measurement were those that are available in source code. Not all of the SPECjvm98 benchmarks' sources are available, because some of them are not publicly available (e. g., the "javac" sources). All in all, 268 classes are loaded by the measurement application.

The results of the class loading measurement are shown in Fig. 4.3. AspectWerkz and JAsCo exhibit the highest overheads: AspectWerkz raises the cost of class loading to about 6.4 times the original cost, JAsCo even to 8.7 times. The operations that are carried out by these environments are, as mentioned above, very expensive.

Reflex benefits from its minimalistic approach, yet the actual cost of using a customised class loader amounts to 28 %. PROSE, on the standard JVM, depending on the debugger infrastructure to intercept class loading, suffers from this, while PROSE/Jikes only has a minimal impact on class loading performance.

Steamloom builds linked lists of instruction objects and indices for join point shadow retrieval at load-time. Especially indexing is a comparatively expensive additional step. This is why Steamloom exhibits a 3.2 times slowdown in class loading as compared to Jikes.

Summary

Generally, it can be concluded that, for long-running applications, executing them in an AOP-enabling environment does not entail unbearable performance penalties. The effects of running an application in such an environment are mainly visible during class loading, which is when most of the approaches perform a considerable amount of their work when no aspects are employed.

The performance of Steamloom when executing SPECjbb2000 was satisfactory when a realistic—with regard to SPECjbb2000 being a *server* benchmark—heap size of 2 GB was used. It must be noted that Steamloom's comparatively weak performance in the presence of a 512 MB heap is due to the mechanisms it employs to support dynamic weaving. BAT instruction lists consume much memory, and this is likely to have a significant impact in a server benchmark when too little memory is available overall. Memory exhaustion due to the employment of BAT is an issue with Steamloom that will be further discussed along with future work directions in Ch. 5.

4.2.3. Core AOP Mechanism Performance

The measurements presented in this section address the following two topics:

- *Cost of executing a join point shadow.* Code at join point shadows can be subject to decoration with advice functionality. It is interesting to measure the cost of the mechanisms employed in different dynamic AOP systems to invoke advice, since measurement results gained thereby form a good foundation for reasoning about the different approaches' efficiency.
- *Cost of reification of join point context information.* This includes, for example, binding advised methods' parameters and passing them to an advice.

The measurements explicitly address all kinds of join point shadows supported in the various systems in scope. To be able to evaluate the performance of the systems, the join point shadows are executed unadvised, to get an impression of the basic performance, and advised, to measure the additional cost due to the advice invocation.

A Suite of Micro-Measurements Such measurements, taking into account the cost of single quasi-atomic operations, are called *micro-measurements*. To provide a set of such micro-benchmarks, a micro-measurement suite was implemented using the JavaGrande benchmark framework [35]. This framework defines three "sections" of benchmark applications, where section 1 comprises micro-measurements (expressing results in operations per second), and sections 2 and 3 contain implementations of core algorithms and complex applications, respectively. Each section provides an infrastructure for building specific benchmark applications. The framework has allowed for implementing, based on the section 1 interfaces, a set of core measurement applications equal for all dynamic AOP systems. For each particular dynamic AOP system, only implementations for the aspects had to be provided, and a small application to set up and deploy them before starting the actual measurement process.

A benchmark based on the JavaGrande section 1 is focused on *one* specific kind of mechanism, e. g., a method invocation or field access. This mechanism is executed a large number of times, until either a certain threshold is reached, or until the time set for the benchmark to run expires. During the execution of the benchmark, the number of executions of the mechanism in question is counted. In the end, the result is computed as the throughput: numbers of mechanism executions per second.

Lst. 4.1 shows the pseudo code of the benchmark method of a micro-measurement based on the JavaGrande framework. All micro-measurements—for method calls, field accesses, etc.—basically follow this structure, where `__op__` in line 6 of the listing represents the operation whose throughput is to be measured. The method `JGFRun()` is the actual measurement method. `MAX_OPS` is the maximal number of operations of this kind that should be executed, and `N` is the number of such operations that should be executed per iteration of the main loop.

```
1  JGFRun() {
2    ops = 0;
3    timer.start();
4    while(ops < MAX_OPS && !timer.thresholdExceeded()) {
5      for(i = 0; i < N; i++)
6        __op__; // see text
7      ops += N;
8    }
9    timer.stop();
10   throughput = ops / timer.timeElapsed();
11 }
```

Listing 4.1: Pseudo code of a JavaGrande-based micro-measurement method.

The functional principle of the micro-measurements for AOP implementations is as follows. For a set of typical join point shadow types, each represented by a different `__op__` in a dedicated measurement application, the benchmark is run in several configurations, and the number of executed join point shadows per second is measured in all of them.

The join point shadow types in focus are these:

- calls and executions of member methods, without and with parameters and return values, and
- read and write accesses to non-static member fields.

Static methods and fields are excluded from the measurements, because they do not deliver new insights about the performance of advice applications.

The configurations for which measurements are made are as follows:

- *Plain:* no advice is attached to the shadow. This measurement yields the basic throughput of the respective shadows in a given AOP environment.
- *Before/after/around advice:* an advice of the given kind is attached to the shadow. The advice's format is, in AspectJ syntax, e. g., `before(): __op__ { ... }`. It

join point shadow	before advice	after advice	around advice
call	target	—	target
call with parameter	argument	—	target, argument
call returning	—	return value	target, return value
call throwing	—	exception	target, exception
execution	this	—	this
execution with parameter	argument	—	this, argument
execution returning	—	return value	this, return value
execution throwing	—	exception	this, exception
field read	target	value	target, value
field write	value	value	target, value

Table 4.1.: Context items accessed in micro-measurements.

does not access join point context in any way. This measurement states how expensive it is in a given AOP environment to attach simple advice invocations to the particular shadows.

- *Context exposure:* the three kinds of advice are attached as before, but this time, a context item is accessed from the advice. Details on which context items are accessed from advice are given in Tab. 4.1.

Because calls to empty methods are too easily optimised away by a modern virtual machine, the various advice that are used in the measurements all simply increment a counter or store passed context information to an internal field. All of the above measurements are executed on all of the systems for which performance measurements are made, to the best possible degree (some systems do not have support for all kinds of advice, for example).

AWBench Apart from the JavaGrande-based suite, that was implemented in the course of developing Steamloom, another micro-benchmark suite called AWBench [23] was used. AWBench is an outcome of the AspectWerkz project [21]. Basically, it follows the same approach; it was incorporated in the evaluation to provide more independent results.

AWBench follows an approach similar to that of the micro-measurements suite introduced above. It is however geared towards measuring the performance of join points that frequently occur in typical applications of AOP. The two measurement suites thus complement each other. The join points for which measurements are performed in AWBench are restricted to *method executions*, since that is the least common denominator of supported join point types in the tools covered by AWBench. To method executions, various types of advice are attached:

- *Before* advice with...
  - no context access (`before`),
  - access to static join point context (`beforeSJP`),

  - access to dynamic join point context (`beforeJP`),
  - access to method arguments and call target (`beforeWithArgsAndTarget`),
  - access to a primitive method argument (`beforeWithPrimitiveArgs`), and
  - access to a reference-type method argument (`beforeWithWrappedArgs`).
- *After returning* advice with access to the returned value (`afterReturning`).
- *After throwing* advice with access to the throws exception (`afterThrowing`).
- A combination of *before and after* advice (`beforeAfter`).
- *Around* advice with...
  - no context access (`around`),
  - access to static join point context (`aroundSJP`), and
  - access to dynamic join point context (`aroundJP`).

After the descriptions, the internal names of the measurements are given. These internal names are the ones by which results are presented below.

For all of these, AWBench performs a constant number of iterations. From the total time measured for each operation, the time needed to execute a single one of each type is computed. Hence, the results of AWBench are expressed in nanoseconds rather than in throughput, as in the JavaGrande-based benchmark introduced above.

Micro-Measurement Results

Not all of the measurements could be performed for all tools. On the one hand, this was due to implementation restrictions of some of the tools, on the other hand, bugs in some of the tools prevented a correct functioning of the measurement applications. In particular, the following implementation restrictions were met:

- JAsCo and Spring AOP only allow for attaching advice to *method executions*. In the case of JAsCo, this is simply a shortcoming of the current state of the run-time weaver. For these tools, no results for any other join point types exist.
- PROSE also only supports method execution join points, in the form of method entry and exit join points. Moreover, it has no around advice. Consequently, the corresponding measurements have not been conducted. PROSE also does not distinguish between before and after advice when the join point in question is a field access. Hence, only before advice were applied for field accesses.
- Steamloom does not support around advice for method calls and executions that throw exceptions. The corresponding measurements were not applied.

The following problem occurred during the measurements: the version of PROSE based on the Jikes RVM crashed when return values from method executions were to be accessed. Also, PROSE/Jikes would not execute advice attached to field get/set operations more than once or twice. When PROSE was run on the Sun VM, these features were functional.

The discussion of micro-measurement results in this section focuses on exemplary characteristics that allow for a good comparison of the regarded systems. Therefore, the results presented here give only those details that are relevant for the discussion. A *complete* collection of figures providing all gathered results is given in the appendix (cf. App. A).

Not all join point types are taken into account. Field access join points have not exhibited any special performance characteristics in the measurements. Moreover, they are not supported by all of the systems. Hence, they are excluded from the discussion.

Method calls and executions that throw exceptions are also excluded, because the performance of these join point shadows is already poor (as compared to non-throwing calls and executions) when no advice are attached. The observation of their performance with attached advice does not yield especially interesting results.

In the following discussion, the focus is on the performance of *method call* and *method execution* join points. Apart from the "plain" case where no advice is attached, all three advice types—*before*, *after*, and *around*—are taken into account. Performance data for *context access* is however only provided for before and after advice, because around advice have not exhibited significant differences between accessing context and not doing so. This is due to the generally higher overhead of around advice, which diminish the impact of additional context access to a non-observable degree. For before advice, the performance when accessing the target object is given; for after advice, the figures show the performance when the return value is accessed.

The results relevant for the discussion are gathered in Fig. 4.4 for call join points, and in Fig. 4.5 for execution join points. In Fig. 4.4, less AOP implementations appear because not all of the regarded systems support call join points. Fig. 4.5 lacks values for PROSE/Jikes in the "after access" case because of the problems mentioned above, and for PROSE in general in the "around" case because PROSE has no support for around advice at method execution join points.

AspectJ exhibits no significant visible cost for before advice in all cases, and for after call advice. For after execution advice, there is an overhead. When context is accessed, additional cost is induced that is significant especially for after call advice. Both of these are *not* observed with AspectJ 1.2 running on Jikes. Given that both the AspectJ 5 and 1.2 compilers generate, apart from advice method names, *exactly the same code* for the given advice types, the observed differences must stem from the VMs' different optimisation behaviours.

JAsCo and AspectWerkz exhibit almost the same performance characteristics. A notable exception are before advice, where JAsCo performs significantly better. The reason for this can be seen from the sequence diagrams shown for the two systems in Figs. 2.6 (p. 62) and 2.5 (p. 57). JAsCo invokes advice directly from application code, while AspectWerkz invokes them through a join point closure, which indirection bears

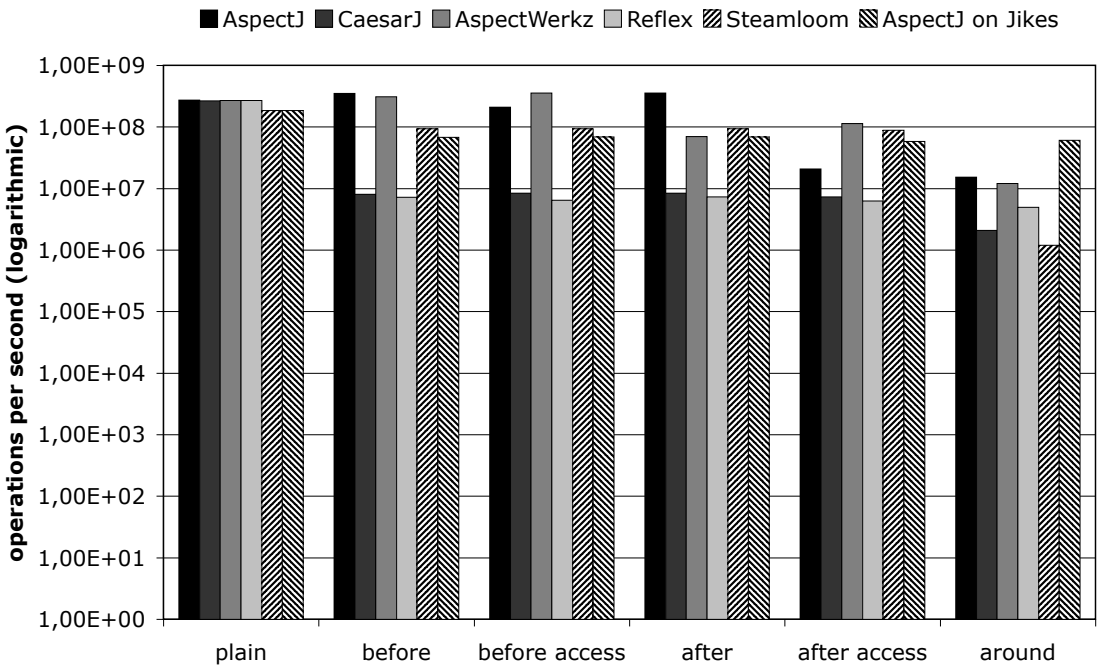

Figure 4.4.: Micro-measurement results for call join points.

some additional cost.

AspectJ and JAsCo exhibit a performance *increase* in some cases. This is presumably due to the shape of woven code, which possibly offers better optimisation opportunities for the Java 5 virtual machine.

CaesarJ and Reflex, which insert a certain amount of infrastructure into the application either at compile-time or load-time, have the same constant overhead for before and after advice: they are about ten times slower than the "plain" case. This is due to the default infrastructure that is introduced. For around advice, CaesarJ performs slightly weaker than Reflex. Its advice are implemented in synchronised methods, which brings additional cost.

Both versions of PROSE exhibit a comparatively weak performance. In both cases, this is due to the complex infrastructure that any advice execution is preceded by. The version of PROSE running on the standard VM, depending on JVMTI breakpoints, additionally suffers from the expensiveness of context switches at such breakpoints. Spring AOP, even though it relies on reflection, performs better than both versions of PROSE. Reflective invocations appear to be less costly than an extensive infrastructure as it is met in PROSE.

Steamloom only exhibits a constant overhead for *all* kinds of advice *regardless* of context access. This is due to the minimalistic approach that the weaver follows for advice block generation. Around advice are an exception: they are much more expensive. Given that around advice proceeding is not supported by the optimising compiler in the present version of Steamloom, this was to be expected.

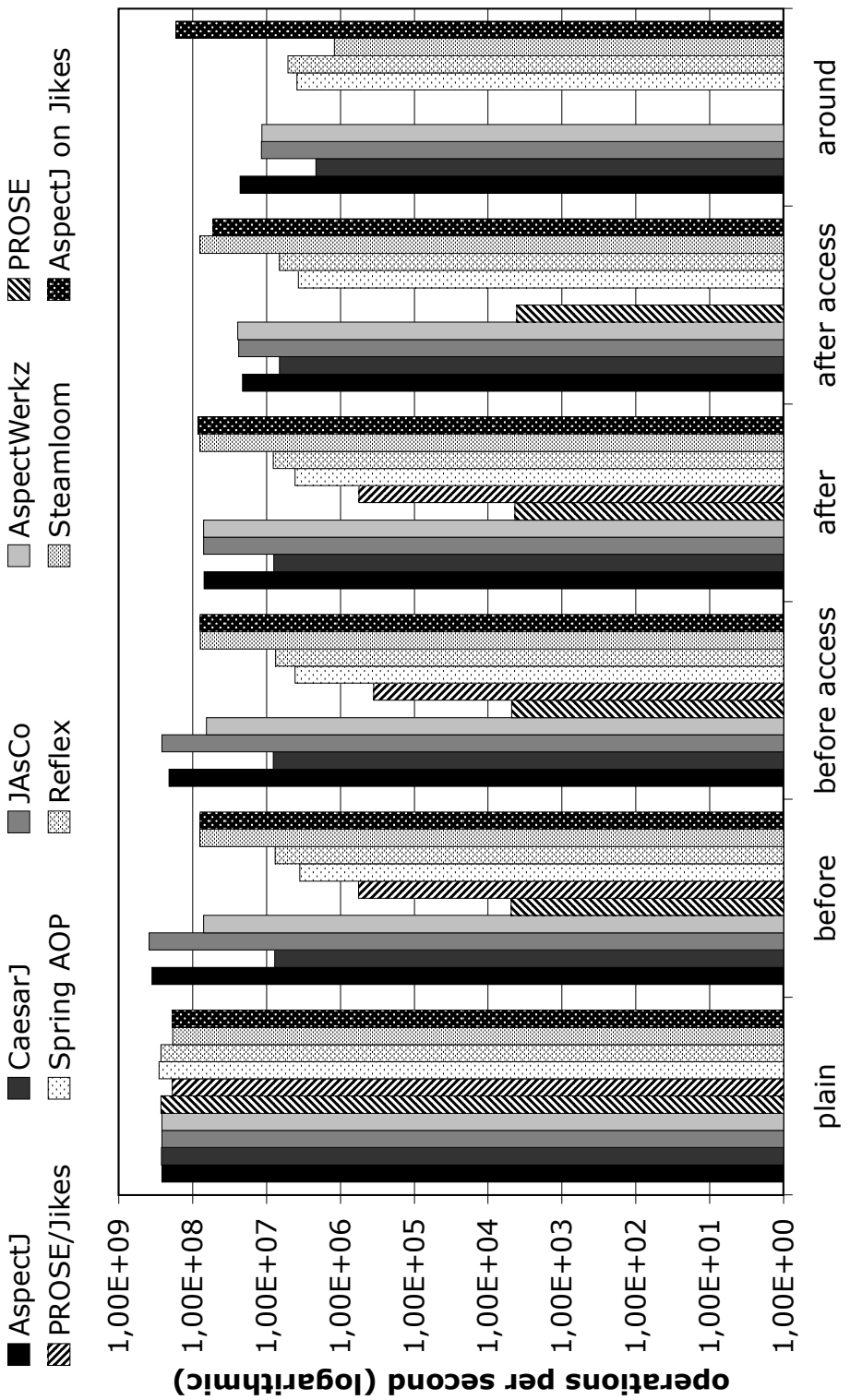

Figure 4.5.: Micro-measurement results for execution join points.

It is especially interesting to compare Steamloom to AspectJ 1.2 running on the Jikes RVM, because the platform is the same in both cases. For call join points, Steamloom generally performs slightly better than AspectJ (apart from around advice, for obvious reasons). For execution advice, both systems exhibit roughly the same performance.

AWBench Results

AWBench was applied to all systems it supports per default, and an additional module for Steamloom was added. The supported systems are AspectJ, AspectWerkz, JAsCo, and Spring AOP. The JAsCo measurements would not run correctly; in fact, the JAsCo compiler—as triggered from the AWBench infrastructure—terminated with an error. Hence, no results for JAsCo are given. The results from the AWBench measurements are gathered in Fig. 4.6. They are grouped by run-time environment to provide a quick overview of the overall cost due to a particular system.

All systems exhibit a strong peak for exception handling. The additional impact on such join points due to advice invocations is negligible. Apart from this peak, which exists because exception handling is generally more expensive than normally terminating method invocations, each of the particular systems exhibits, in general, a levelled profile. There are some exceptions that will be explained in the following.

AspectJ and AspectWerkz have high costs for access to dynamic context. AWBench requires the aspect under measurement to retrieve the target of a method execution from the join point context. In the AWBench implementation, both AspectJ and AspectWerkz access, to achieve this goal, an instance representing the join point that is passed to the advice. In the case of AspectJ, this is done via `thisJoinPoint`, and in AspectWerkz, the advice is passed a `JoinPoint` instance. These instances have to be created dynamically, which is the reason for the high cost.

It may be argued that, to simply access the target of a method execution, much more efficient means are available in AspectJ and AspectWerkz: the target could be bound using the `target` pointcut designator. Still, it appears to be a requirement to access the reified join point context. Steamloom does not support reifying the dynamic join point context as a whole. Instead, single items from the context can be passed to advice (cf. Sec. 3.6.4). This particular part of the benchmark was implemented by sending the advice the `appendTarget()` method during configuration. Hence, Steamloom does not exhibit significant additional cost.

In AspectWerkz, static context access is much more expensive than in AspectJ. This is because AspectJ can resolve parts of the static join point context—such as, for a method call join point, the method being called—at compile-time and make them implicitly available. AspectWerkz advice accessing static join point context accept a `StaticJoinPoint` object that is set up in the same fashion as a join point object representing the dynamic context, only that it contains less information. In any case, the `StaticJoinPoint` object is assembled *at run-time*, which induces the observed cost.

Around advice exhibit very different costs in AspectJ and AspectWerkz. The reason for this is that AspectJ generally follows an inlining strategy when weaving around advice, while AspectWerkz invokes around advice just like other advice: they are rep-

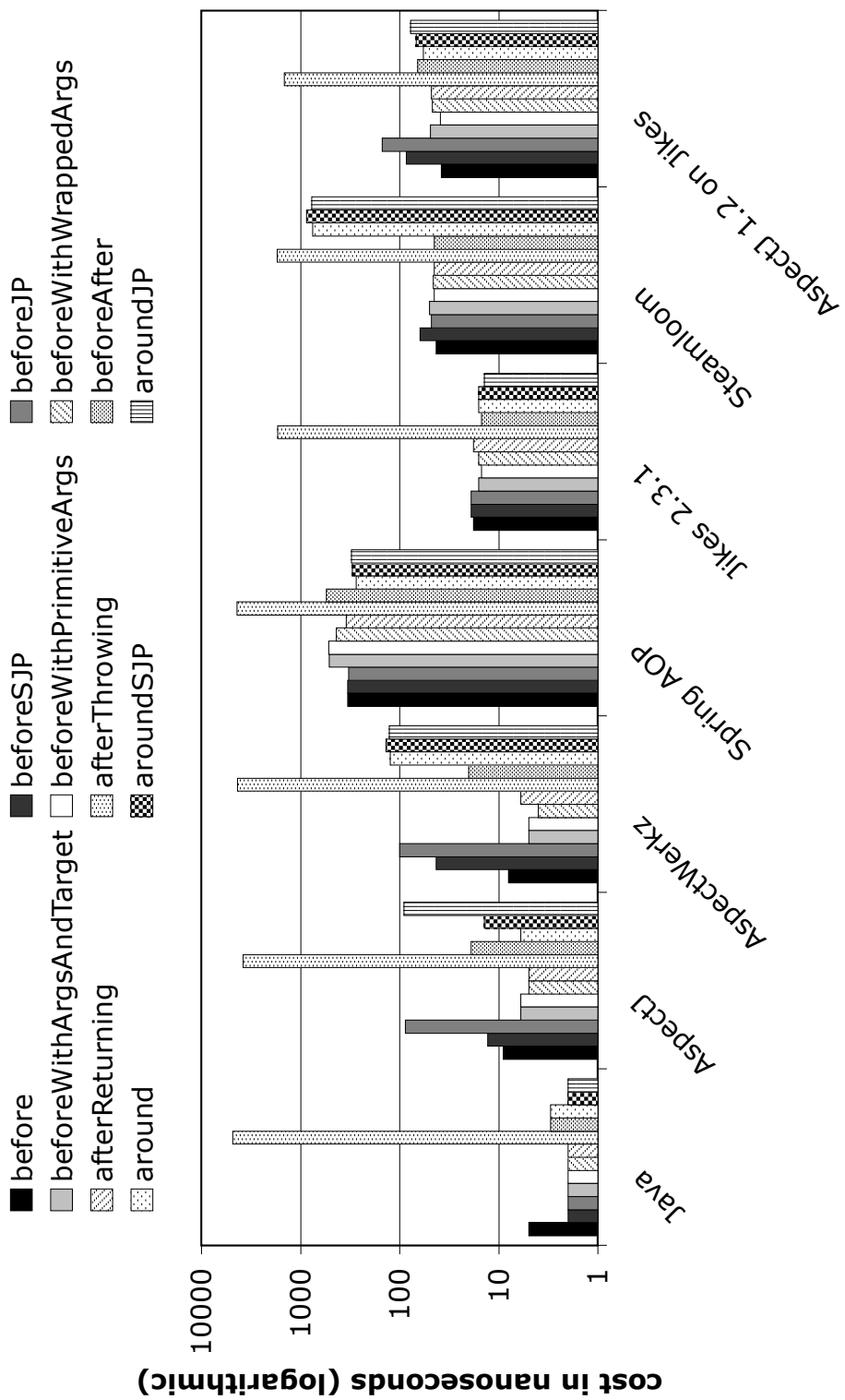

Figure 4.6.: Results from AWBench measurements.

resented as methods but retrieve an instance representing the join point, to which they can send the `proceed()` message.

Spring AOP, relying on reflection to invoke advice, is, as expected, clearly expensive in all cases.

Because the Jikes RVM is generally a little slower at invoking methods than the standard JVM, Steamloom's cost in executing advice is also higher than that of most of the systems running on the standard JVM. Still, it performs reasonably well when compared to AspectJ running on Jikes. Steamloom is actually faster at accessing static and dynamic join point context. On the one hand, it benefits from its minimalistic weaving approach for dynamic join point context access, where the weaver always generates exactly those bytecode instructions needed to access the requested value (and where the efficient `peek` bytecode can be used). On the other hand, Steamloom benefits from VM-internal storage of static join point context information.

Steamloom's around advice are not expected to perform extremely well, since their implementation is not supported by the optimising compiler.

Summary

From the results presented above, it can be concluded that the implementation approach followed by Steamloom is indeed beneficial. Steamloom does performs very well—except for around advice—when compared to AspectJ running on Jikes. In the case of around advice, Steamloom's weaker performance is due to the limited implementation, which does not support optimised compilation of around advice. Steamloom can even bear comparison with systems like AspectWerkz in some cases. Steamloom's approach to dynamic weaving is clearly superior to all other dynamic weaving approaches, especially to those that operate at the meta-level or incorporate hooks and wrappers. Its minimalistic bytecode weaving strategy implies minimal overheads.

All in all, Steamloom exhibits very good performance while providing *fully dynamic* weaving. The comparison with AspectJ 1.2 running on Jikes is interesting in this regard: code woven by Steamloom does not perform significantly worse, but weaving *does* take place in a fully dynamic way. An implementation approach following the idea of tight integration of AOP functionality with an actual execution layer thus appears to be a good solution to address.

4.2.4. Footprint of Deactivated Aspects

Some AOP implementations allow for dynamic weaving, i. e., for weaving aspects in and out while the base application is running. Some of the AOP systems with dynamic weaving capabilities perform preparation steps for dynamic weaving when the application is loaded into the run-time environment. During the preparation phase, no advice are attached to join point shadows, but the shadows are modified in a way that allows for later attachment of advice by the AOP environment.

The preparation frequently consists of either *wrapping* the respective join point shadows in calls to the AOP infrastructure, or of adding *hooks* before and after the shadows

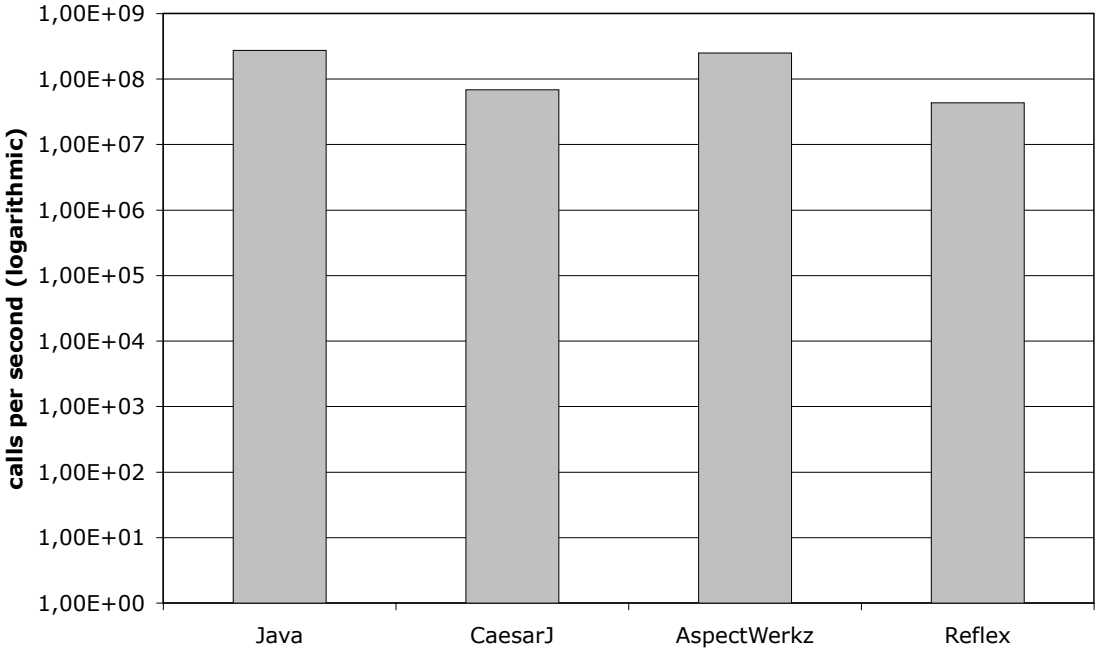

Figure 4.7.: Results of footprint measurements.

that also call the infrastructure. Both hooks and wrappers act differently based on the shadow's being affected by advice or not.

The analysis in this section is based on a simple benchmark measuring the cost of executing join point shadows that have been prepared for advice attachment, but that have *not* been attached advice. The measurement is based on the micro-measurements suite introduced in Sec. 4.2.3; it is restricted to method call join points. The only systems regarded are CaesarJ, AspectWerkz, and Reflex. These three systems all modify base application code at compile-time or load-time solely based on the *possibility* of aspects being deployed later. None of the other systems effect a presence of inactive aspects in base application code.

Measurement results for plain Java and the three aforementioned AOP implementations are presented in Fig. 4.7. From the figure, it can be seen that CaesarJ and Reflex both impose a certain overhead on a running application even when join point shadows are merely *prepared* for advice attachment. Conversely, AspectWerkz does not exhibit a significant cost.

Figs. 4.8, 4.9, and 4.10 show sequence diagrams for the three systems that are based on the diagrams presented for these systems in their corresponding sections in Ch. 2. The diagrams here show the control flows that are associated with the execution of join point shadows when no advice are attached.

AspectWerkz merely introduces a simple indirection into the execution of method calls, hence its minimal footprint. When looking at the control flows of CaesarJ and Reflex, it is evident why CaesarJ performs better: less objects and method calls are

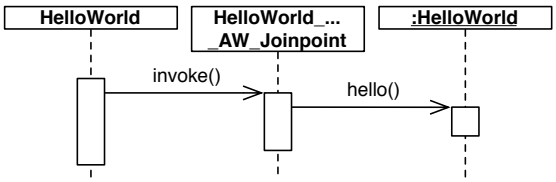

Figure 4.8.: Control flow at a prepared join point shadow in AspectWerkz.

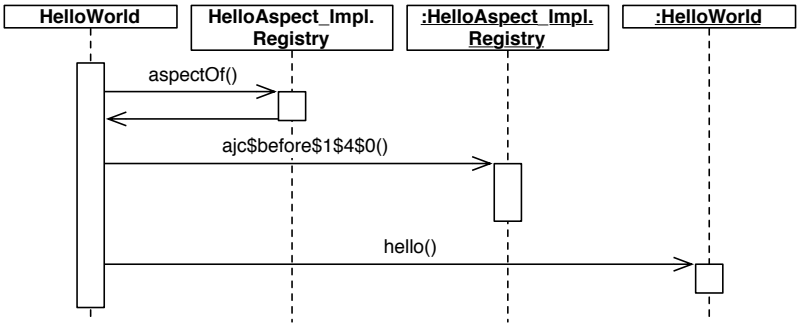

Figure 4.9.: Footprint of a deactivated before call advice in CaesarJ.

involved than Reflex uses. As for the actual cost imposed on method calls, they are roughly 10 % more expensive in AspectWerkz than they are in plain Java. In CaesarJ, they are 4 times as expensive; in Reflex, 6.3 times.

It can be concluded that most of the systems regarded in the evaluation leave *no footprint* in base application code when aspects are deactivated. AspectWerkz leaves a very small footprint: some amount of infrastructure is executed, but it is very minimal and does not impose a significant overhead on the execution of the base application. Reflex and CaesarJ both have a considerable footprint, executing conditional logic and querying data structures for advice presence, and thereby raising the cost of executing join point shadows.

4.2.5. Performance of Scoped Aspects

In the context of this work, *scoping* addresses the restriction of aspects' areas of validity to single threads or instances. As mentioned in Sec. 3.10, Steamloom does not support constructs like `perthis`, `pertarget`, or `percflow`. Hence, the latter are not regarded in the scoping measurements.

The measurements that are conducted in this section focus on the regarded AOP implementations' capabilities of restricting the applicability of aspects to single objects or threads. The measurements are run using the micro-benchmark suite introduced in the previous section. This time, the only kind of join point shadows regarded are method executions. Basically, this means that the cost of executing a method on a given object is measured for the following two cases:

- *Out of scope:* the aspect in question is not valid in the current thread or has not

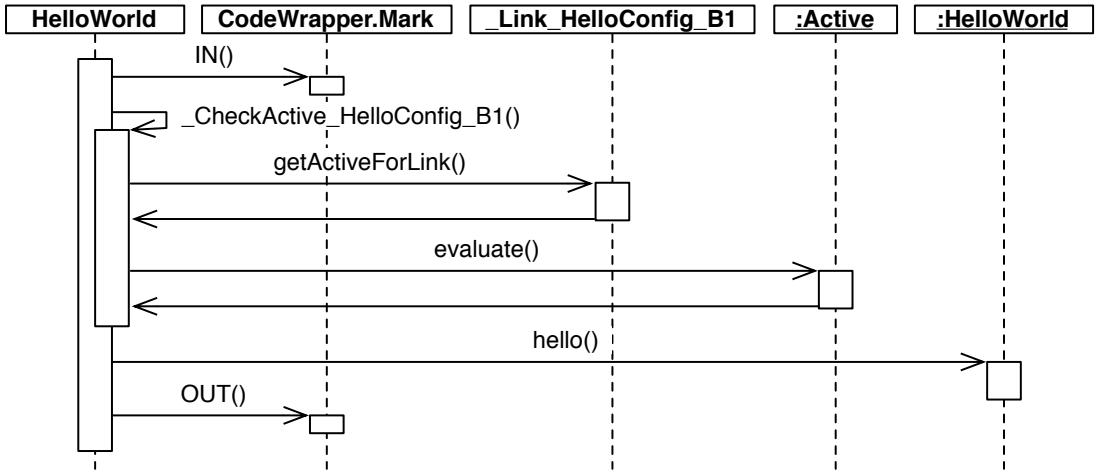

Figure 4.10.: The footprint of a deactivated link in Reflex.

been deployed on the object that executes the method.

- *In scope:* the aspect has been deployed on the current thread or object, respectively.

A simple before advice is attached to the join point shadow for both thread-local and instance-local deployment.

AWBench has no support for measuring scoping, so it is not used in this context.

Steamloom supports scoping both for instances and threads. Of the other systems, JAsCo, PROSE, and Spring AOP have direct support for instance-local aspects. Thread-local deployment is only directly supported by CaesarJ. Details on this can be found in the respective sections in Ch. 2. For all systems that do not have direct support for a scoping mechanism, the mechanism was emulated. The emulation of instance-local and thread-local deployment shall now be explained by example of AspectJ.

Emulating Instance-Local Deployment Instance-local aspect deployment can be emulated in AspectJ as follows. Lst. 4.2 shows an aspect that executes an advice before every execution of a method `C.m()`. The aspect decorates single objects or groups thereof, that have to be stored in the internal hash set through the aspect's `addInstance()` method. This approach naturally imposes a certain overhead on advice execution, as the set containment check has to be done at every matching join point. Moreover, aspect code is executed regardless of whether the advice will be actually invoked or not.

Apart from Steamloom, the systems that have support for instance-local deployment are PROSE, JAsCo, Reflex, and Spring AOP. The emulation approach has been implemented for all other systems.

Emulating Thread-Local Deployment Thread-local deployment can be emulated very much in the same way as instance-local deployment. Basically, the approach in Lst. 4.2

```
1  public aspect InstanceLocal {
2    private static Set instances = new HashSet();
3    public static void addInstance(C c) {
4      instances.add(c);
5    }
6
7    pointcut affected(C c) : target(c) && if(instances.contains(c));
8
9    before(C c) : execution(public void C.m()) && affected(c) {
10     // advice functionality
11   }
12 }
```

Listing 4.2: Instance-local aspects in AspectJ.

can be reused. The emulation is shown in Lst. 4.3. As with the instance-local emulation, the thread-local emulation was implemented for all systems that do not directly support such scoping. Steamloom and CaesarJ are the only systems that support thread-local deployment.

```
1  public aspect ThreadLocal {
2    private static Set threads = new HashSet();
3    public static void addthread(Thread t) {
4      threads.add(t);
5    }
6
7    pointcut affected(): if(threads.contains(Thread.currentThread()));
8
9    before() : execution(public void C.m()) && affected() {
10     // advice functionality
11   }
12 }
```

Listing 4.3: Thread-local aspects in AspectJ.

Results The measurements succeeded for all systems but the two versions of PROSE, which crashed when its instance-local deployment capabilities were used, and which did not execute the thread locality-emulating advice at all for unknown reasons[2].

Results from the micro-measurements are gathered in Fig. 4.11. Apart from the method execution throughput for unaffected and affected instances and threads, the throughput for completely unadvised method executions is given as the "plain" value to provide a relation.

All systems exhibit certain performance penalties when scoped aspects are employed. AspectJ, AspectWerkz and Reflex behave as expected with regard to the results gained and discussed for advice invocations at execution join point shadows in Sec. 4.2.3: the results show that a certain price must be paid for residues that check set containment.

[2]Attempts to retrieve detailed information from the PROSE developers were not successful.

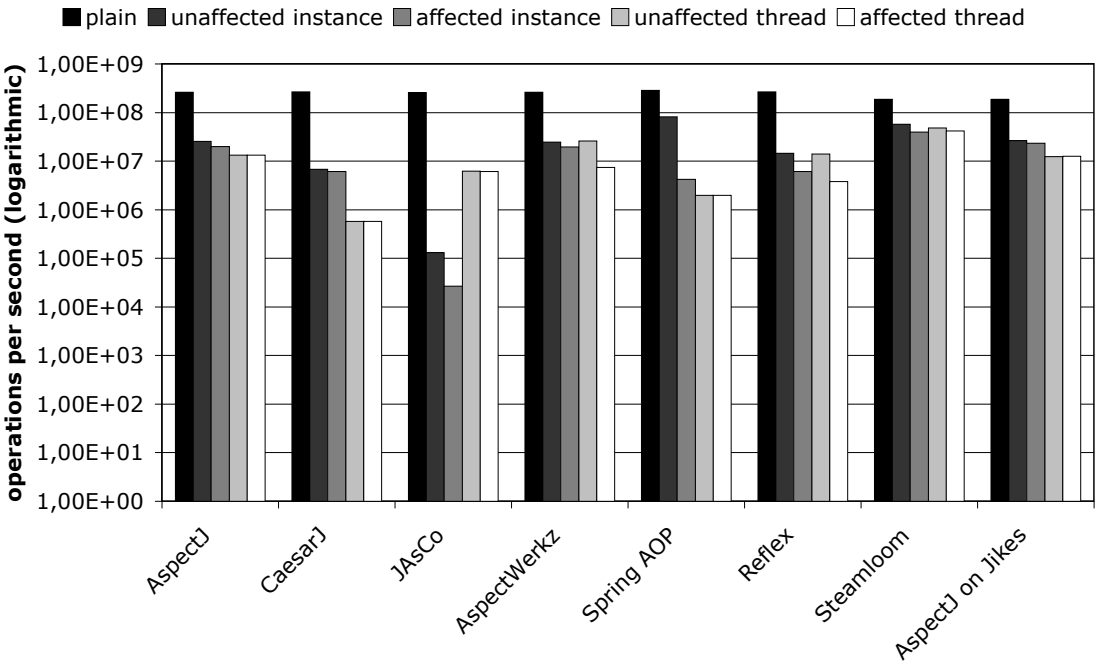

Figure 4.11.: Measurement results for instance-local and thread-local aspects.

These systems will not be discussed below; instead, those AOP implementations that exhibit especially interesting results are dealt with.

In JAsCo, the performance downgrade is especially evident because the run-time weaver, as mentioned in Sec. 2.5, does not support instance-local deployment but reverts to an alternative implementation strategy that is much slower. Thread-local deployment in JAsCo exhibits results that were to be expected given the results for method executions presented in Sec. 4.2.3.

Spring AOP performs best for unaffected instances. This is because its proxies are only attached to instances that are actually decorated. Thread-local deployment is notably more expensive because the advice must be executed in all cases to check the thread applicability.

CaesarJ, being the only of the systems (apart from Steamloom) that directly supports thread-local deployment in its programming model, suffers from the additional complexity that is induced by its implementation. For thread-local advice, CaesarJ performs more complex lookups of advice instances via hash tables.

Of all systems, Steamloom exhibits the best performance (apart from unaffected instances in Spring AOP). The downgrades observed are due to residue overhead in the case of thread-local deployment, and due to inlining prohibition in the case of instance-local deployment. However, the fact that both deployment approaches have *dedicated* support from the VM *itself* increases performance. These results show once more that VM integration must be adopted if language mechanisms are to be supported efficiently.

4.2.6. Performance of `cflow`

For `cflow`, three different measurement approaches were taken. The micro-measurement suite introduced in Sec. 4.2.3 was used. Moreover, two measurement applications were used that deliver more detailed data on the three `cflow` weaving strategies implemented in Steamloom (cf. Sec. 3.11).

Micro-Measurement Suite The micro-measurement suite is used in the standard way for method call join point shadows. As in the measurements on scoping discussed in the previous section, the join point shadows are executed in two different contexts, where an advice depending on the control flow is attached to them:

- *Out of `cflow`:* the measurement method is called from outside the control flow.
- *Inside `cflow`:* the measurement method is called from within the control flow.

In Lst. 4.4, the pseudo code of this benchmark can be seen. The code constituting the `JGFRun()` method was omitted; it is the same as in Lst. 4.1. In this case, there are *two* measurement methods instead of one: `CFlow()` and `JGFRun()`, where the former simply calls the latter. The aspect employed in this case has an advice of the form (in AspectJ syntax), e. g., `before(): __op__ && cflow(call(void CFlow()))`. In these measurements, only the performance of before advice attached to method calls and executions was regarded.

```
1 void JGFRun() {
2   ...
3 }
4 void CFlow() {
5   JGFRun();
6 }
```

Listing 4.4: Pseudo code of a JavaGrande-based micro-measurement for `cflow`.

The measurements give an impression of the throughput an AOP implementation yields when a dependent join point shadow occurs inside or outside a control flow. In case of Steamloom, all three implemented `cflow` weaving strategies are measured.

These measurements are, however, not sufficient to analyse how the different `cflow` approaches scale that have been implemented in Steamloom. To better evaluate these, another two measurement applications were implemented and applied to the various AOP implementations.

Variability Benchmark The purpose of this benchmark application is to measure the performance of the introduced `cflow` implementations. This is done with special regard to how performance varies depending on the number of control flow entries/exits and dependent join point shadow occurrences in- and outside of the control flow. In addition to that, it measures how well `cflow` implementations scale with an increasing number of threads.

```
public class CflowBenchmark {
  int entries, deps;

  public void test(
    int outer, int freq_cflow, int inner,
    int freq_dep
  ) {
    while (outer-- != 0) {
      int real_inner =
        (outer % freq_dep == 0) ? inner : 0;
      if (outer % freq_cflow == 0) {
        m(real_inner);
      } else {
        m0(real_inner);
      }
      Thread.yield();
    }
  }

  public void m(int runs) {
    entries++;
    while (runs-- != 0) {
      foo();
      x();
    }
  }

  public void m0(int runs) {
    while (runs-- != 0) {
      foo();
      x();
    }
  }

  public void foo() {
    x();
  }

  public void x() {
    ++deps;
  }
}
```

Listing 4.5: Source code of the `cflow` benchmark.

```
aspect Counting {
  before():
    cflow(execution(* CflowBenchmark.m(int))) &&
    call(* CflowBenchmark.x()) {
      CFlow.count();
  }
}
```

Listing 4.6: Aspect for the `cflow` iteration benchmark.

The source code of the core measurement class is shown in Lst. 4.5. The aspect applied to the benchmark class is shown in Lst. 4.6. As can be seen from the latter listing, the control flow entries and exits in the benchmark are defined by executions of the method `CflowBenchmark.m()`. The dependent join point shadows are calls to `CflowBenchmark.x()`.

The benchmark basically consists of two nested loops. The outer loop controls the number of control flow entries and exits encountered during a benchmark run. It is implemented in `CflowBenchmark.test()`. The inner loop controls the number of executions of the dependent join point shadow. It is implemented in `CflowBenchmark.m()` and `m0()`, respectively, for reasons that will be described below.

The `CflowBenchmark.test()` method is the entry into the benchmark. It accepts four parameters, each of which controls a certain facet of the benchmark behaviour:

- `outer` denotes the number of iterations to be performed by the outer loop. The additional parameter `freq_cflow` defines, in terms of the fraction of iterations, how many of the iterations actually enter and exit the control flow.

- `inner` controls the number of iterations in the inner loop. The actual number of executions of the dependent join point shadow can be controlled by setting the `freq_dep` parameter. It defines the fraction of control flows in whose context a dependent shadow is actually reached.

Fig. 4.12 shows an abstracted sample run of the measurement in one thread. The values of the different parameters are given in the figure. Where an iteration of the outer loop enters the `cflow`, the corresponding box representing that iteration is marked grey. Since `freq_cflow` is set to 2 in the example, every second iteration actually enters the control flow. The `freq_dep` parameter is set to 3. Hence, every third iteration of the outer loop leads to actual executions of the inner loop, and thereby to executions of the dependent shadow. Where the latter are executed inside the control flow, the boxes representing them are also marked grey.

The benchmarking functionality in the `CflowBenchmark` class is run from a harness application that reads, from the command line, values for the four parameters passed to `CflowBenchmark.test()`, and the number of threads in which the benchmark should be run. It then instantiates the desired number of threads and invokes the benchmark in each of them. The *entire* benchmark is run twice, where the second run is used to generate benchmark results. The first run is for warming up the virtual machine, i. e., for bringing the benchmark application to its "steady state" where the virtual machine has already applied optimisations. This serves the purpose of actually measuring the intended performance characteristics instead of the overhead the virtual machine imposes due to performing optimisations. The result of the benchmark is the time the environment needed to run the benchmark.

Nested Control Flows Benchmark The purpose of this benchmark is to measure the performance of `cflow` implementations when nested `cflow` statements are used. In

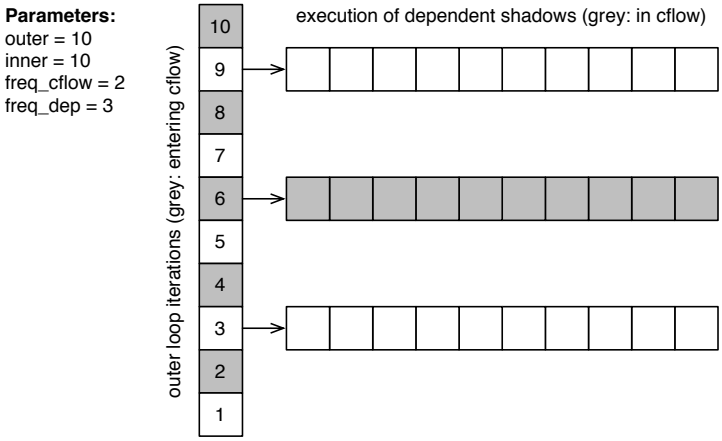

Figure 4.12.: Execution of control flow entries/exits and dependent shadows in the `cflow` measurements.

the benchmark, ten methods `f0()`–`f9()` invoke each other *recursively* in all possible permutations. The last method in the row always invokes a method `foo()`.

The benchmark applies an aspect with a pointcut/advice combination like the one in Lst. 4.7 to the application. It nests ten `cflow` pointcuts in some order and attaches an advice to the execution of the `foo()` method.

```
1  before():
2    cflow(execution(void X.f0()) &&
3      cflow(execution(void X.f1()) &&
4        cflow(execution(void X.f2()) &&
5          cflow(execution(void X.f3()) &&
6            cflow(execution(void X.f4()) &&
7              cflow(execution(void X.f5()) &&
8                cflow(execution(void X.f6()) &&
9                  cflow(execution(void X.f7()) &&
10                   cflow(execution(void X.f8()) &&
11                     cflow(execution(void X.f9()))))))))))) &&
12   execution(void X.foo()) {
13     X.count++;
14   }
```

Listing 4.7: An aspect with nested `cflow` pointcuts.

The semantics of this aspect is that the advice will be executed *only* if the ten `f...()` methods are on the call stack in *exactly* the order determined by the nested `cflow` designators. That is, the advice will, during the benchmark, be executed exactly once, when the correct permutation of `f...()` methods is on the stack.

This benchmark measures the time the application takes to run. It yields information on the cost of entering and leaving control flows, and also on the cost of checking `cflow` matches when nested control flows are on hand.

Micro-Measurement Results

The simple `cflow` benchmarks were applied to all systems in focus. The results from these measurements are shown in Fig. 4.13. For JAsCo, PROSE, and Spring AOP, no results are given for method call join points because these systems have no support them.

It is immediately visible that the approaches using stack walking perform worst. Still, the stack walking implementation based on Steamloom performs considerably better than JAsCo, PROSE, and Spring AOP, which create `Throwable` instances to access the call stack. Inside the virtual machine, the call stack is immediately available for access, while a representation must be expensively created when it is accessed at application level.

Even though stack walking clearly benefits from an integration into the VM, it still performs significantly worse than *all* other approaches. The conceptual benefit of the stack walking approach—it needs residues *only* at dependent join point shadows—is annulled by the high cost of the residues.

The counter-based approaches all perform better. The differences they exhibit stem from implementation details whose effects on advice invocations are basically the same as have been explained in Sec. 4.2.3. Small additional overheads are due to counter management and condition checks.

Steamloom's counter implementation is the most efficient of all counter-based approaches. It even outperforms AspectJ running on the standard VM. Basically, both approaches use thread-local counters to monitor control flows, but AspectJ does so at application level using `ThreadLocal` instances (cf. 2.2, while Steamloom directly associates the counters with the VM's internal representation of threads. Residues checking the counters also are immediate calls *into* the VM in Steamloom, while AspectJ executes all checks at application level, expressed in bytecode.

Continuous weaving performs extremely well. Given the nature of this benchmark—the control flow is only ever entered and left once—, this was to be expected; this benchmark is not suited to yield accurate results concerning continuous weaving. The benchmark results described in the following subsection will give better insights on its performance, since they enter and leave control flows more ofter.

Variability Benchmark Results

The more aggressive benchmarks were not applied to all systems. Those systems that exhibited extremely weak performance in the simple `cflow` benchmark, namely JAsCo, PROSE, and Spring AOP, were excluded. This was done because from the variability and nesting benchmarks that explicitly address scaling and complex nested control flows, no new insights with regard to these systems' `cflow` implementations were to be expected.

The *variability benchmark* was run in a large number of different configurations on all remaining systems. In fact, it was run for *each* combination of the following values:

- 1, 5, 10, 15, and 20 threads,
- 1000 outer and inner iterations,

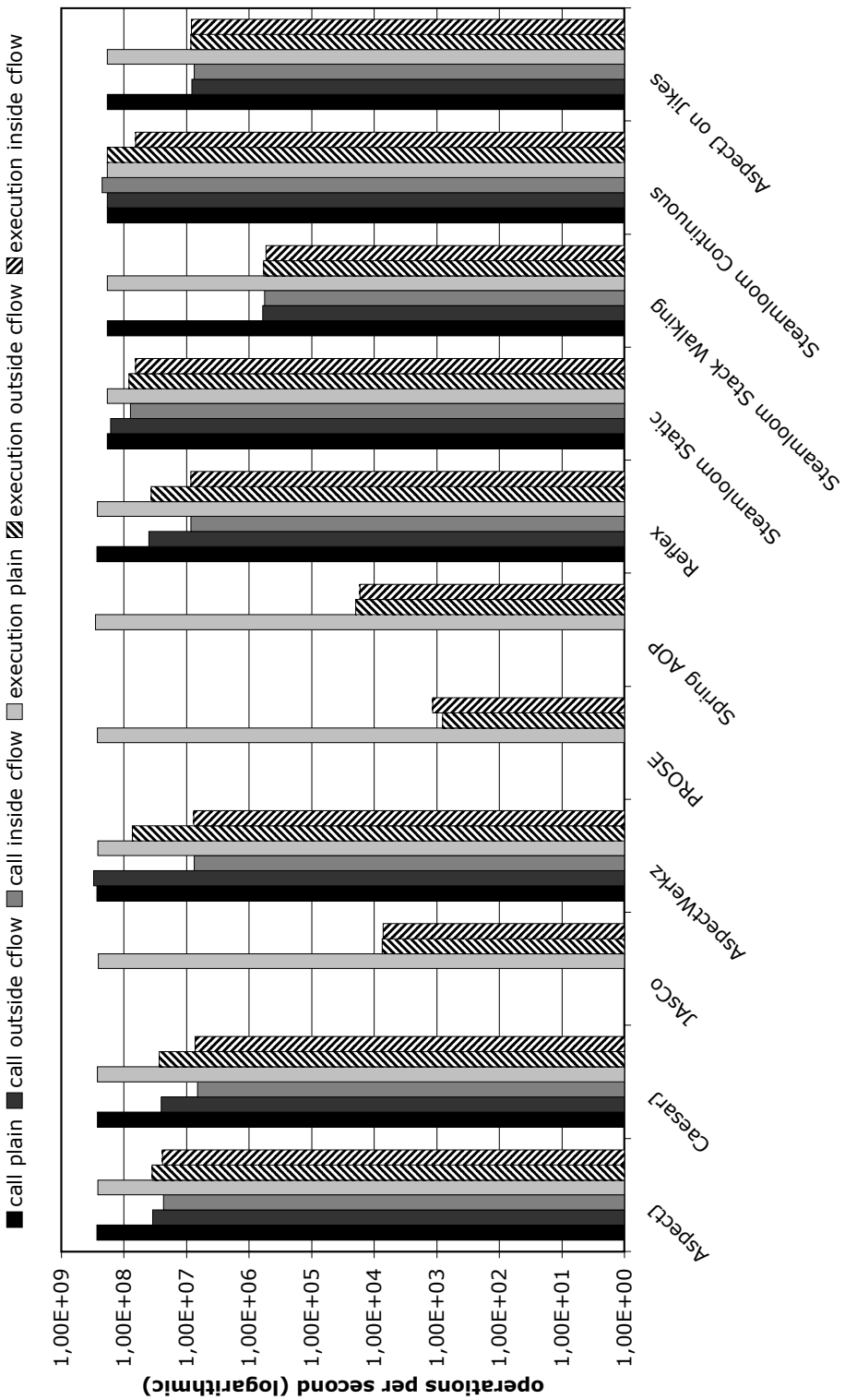

Figure 4.13.: Results from `cflow` micro-measurements.

- 1, 2, 5, 10, 100, and 1000 as the fraction of outer iterations entering the control flow, and
- 1, 2, 5, 10, 100, and 1000 as the fraction of inner iterations executing the dependent shadow.

Applying all combinations of these values to all systems has yielded a vast amount of raw data that will not be presented as a whole here. Rather, the discussion focuses on some points of special interest that exhibit characteristic behaviour. A measurement point is defined by a triple (t, c, d), where t denotes the number of threads, c denotes `freq_cflow`, and d denotes `freq_dep`.

The measurement points discussed below are as follows:

1. $(1, 5, 100), (10, 5, 100), (20, 5, 100)$: these measurement points often enter the control flow (in a fifth of all outer iterations), but seldom execute the dependent shadow (in a hundredth of the inner iterations).
2. $(1, 100, 5), (10, 100, 5), (20, 100, 5)$: these exhibit the inverse behaviour, entering the control flow infrequently but reaching the dependent shadow more often.
3. $(1, 5, 5), (10, 5, 5), (20, 5, 5)$: both control flow and dependent shadow are frequently executed.
4. $(1, 100, 100), (10, 100, 100), (20, 100, 100)$: both control flow and dependent shadow are infrequently executed.

For each of the measurement points, a separate figure is given containing the values for all approaches that have been measured (see Figs. 4.14–4.17).

It is immediately evident that continuous weaving is a prohibitively expensive approach when control flows are entered and left frequently. The only scenario in which it ranges in the average is the second measurement point set, where the frequency of dependent shadows is significantly higher than that of control flow occurrences. This was to be expected: recompilation is an expensive operation, and when it needs to be frequently done, the overall performance of the system suffers.

Stack walking is extremely expensive exactly in the scenario where continuous weaving performance is average: when many dependent shadows are executed, but when this seldom happens in a control flow. This was also to be expected: when the control flow is not active, the entire stack must be walked to yield a negative result. Matching is done much faster when it is successful, as in the first scenario.

Continuous weaving and stack walking are suited for opposite scenarios with extreme differences between the frequencies of control flows and dependent shadows. This can also be seen from the results for the two gathered in the third and fourth scenario, where control flows and dependent shadows both occur at the same frequency.

Counter-based approaches generally yield the best performance. Differences lie in the particular implementations of counter storage and management. The effects of different implementation approaches become especially visible when the first two scenarios are regarded.

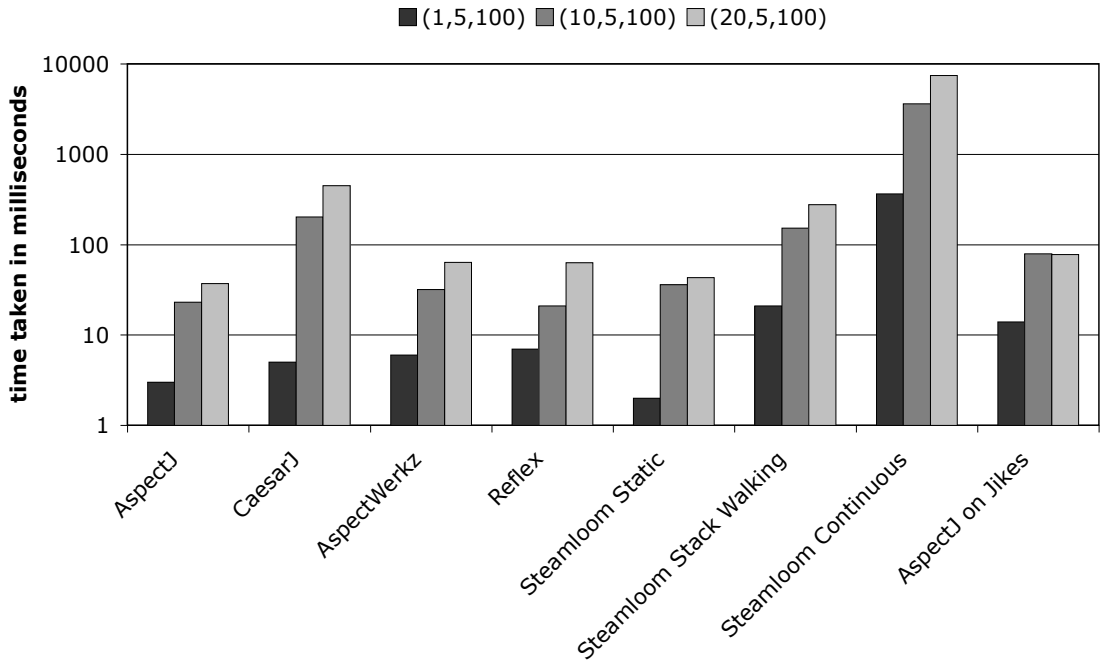

Figure 4.14.: Variability benchmark results for the first measurement point set (frequent control flow, infrequent dependent shadow).

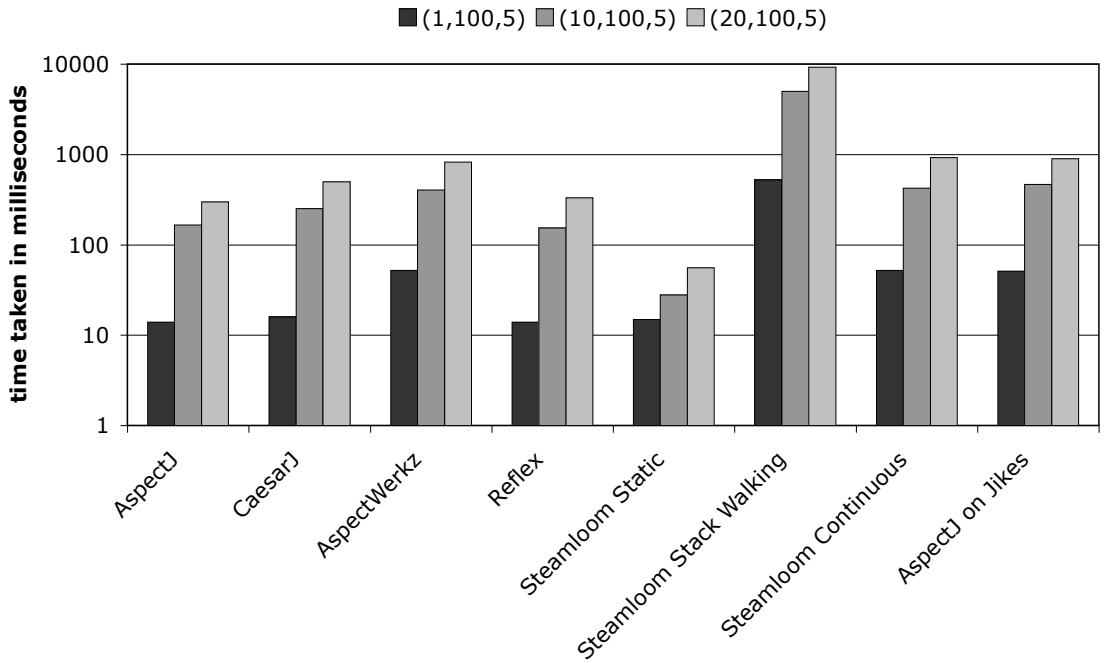

Figure 4.15.: Variability benchmark results for the second measurement point set (infrequent control flow, frequent dependent shadow).

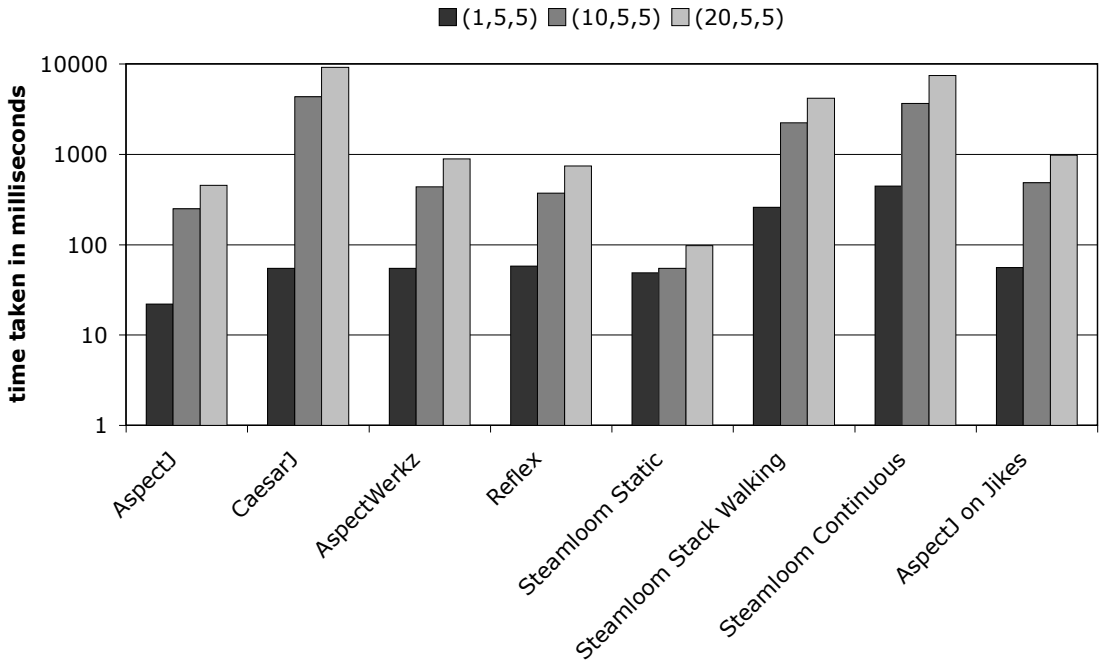

Figure 4.16.: Variability benchmark results for the third measurement point set (frequent control flow and dependent shadow).

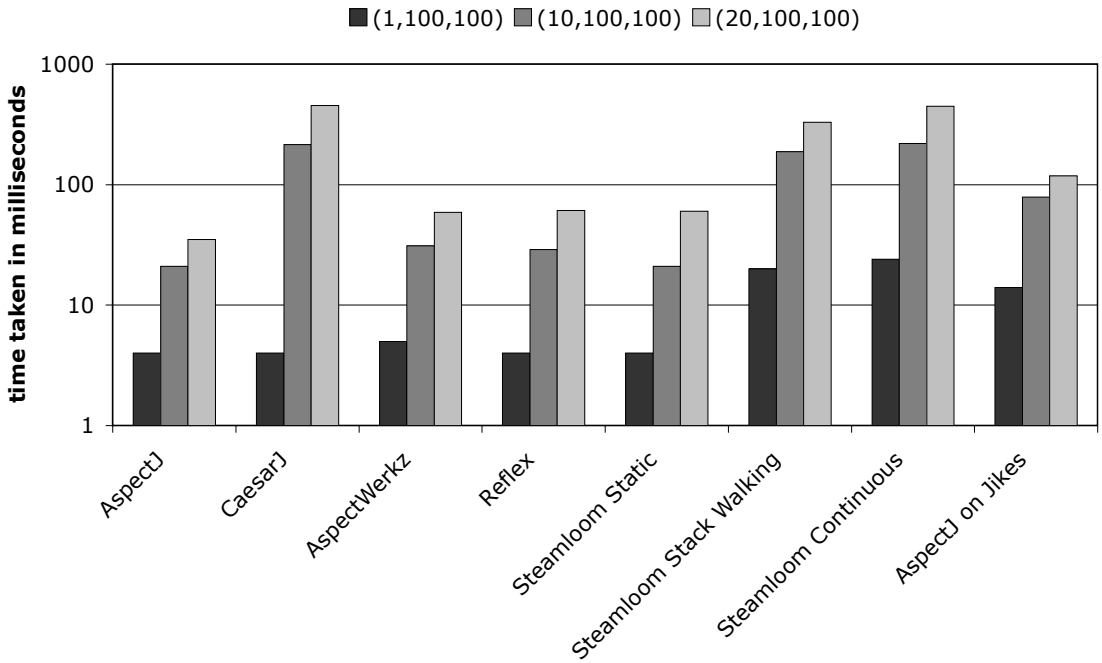

Figure 4.17.: Variability benchmark results for the fourth measurement point set (infrequent control flow and dependent shadow).

It is a characteristic of the first scenario that the operation that is most frequently executed in it is control flow counter update. Conversely, the second scenario's characteristic is that the most frequent operation is counter check. Thus, the performance of systems in the first scenario is an indicator for the efficiency of counter management for updating, and the performance in the second scenario indicates the efficiency of counter checks.

Of the counter approaches, Steamloom obviously provides the most efficient implementation for counter updates, maintaining counters directly in the VM's internal representation of threads. It is still among the best when it comes to checking the counter.

AspectWerkz reveals a weakness in counter check efficiency. This due to its implementation approach; it does not merely encapsulate counters, but maintains stack objects, and a control flow check needs to determine whether the size of the stack is larger than zero. This operation is more expensive than a simple arithmetic comparison on numbers.

A very interesting aspect with regard to the efficiency of `cflow`-related operations is their scaling behaviour when multiple threads are involved. In this respect, Steamloom's integrated approach once more reveals its benefits: it generally is among the most moderately scaling systems and never performs worst. Especially when a large number of threads often executes dependent shadows to which only few control flows apply, its well-scaling counter check pays off.

CaesarJ exhibits the most conspicuous scaling behaviour, which is due to it not using `ThreadLocal` instances or similar concepts, but hashing data structures mapping thread instances to other objects.

Nested Control Flows Benchmark Results

Results for the nested control flow benchmark are shown in Fig. 4.18. This measurement was not applied to the continuous weaving approach of Steamloom, because the extremely high frequency of entering and leaving control flows, which has already led to bad results in the previous benchmark, suggests that its performance is extremely weak in such circumstances[3].

The results show once more that the stack walking approach which has already proven to be suboptimal in the simple control flow benchmarks is indeed not optimal, even if support for it is integrated in the virtual machine. The cost of walking the stack is too high, and non-matching stacks impose a high cost on the matcher.

All other approaches that use counters perform better. AspectWerkz suffers from its expensive counter management strategy. Steamloom once more benefits from the integration of counter storage with VM-internal data structures. It even performs better than some of the approaches based on the Sun standard VM. Put in relation to AspectJ running on Jikes, it becomes obvious that Steamloom's approach is beneficial.

[3] In fact, the benchmark was, as an experiment, run on the continuous weaving implementation; the process was terminated after 20 minutes.

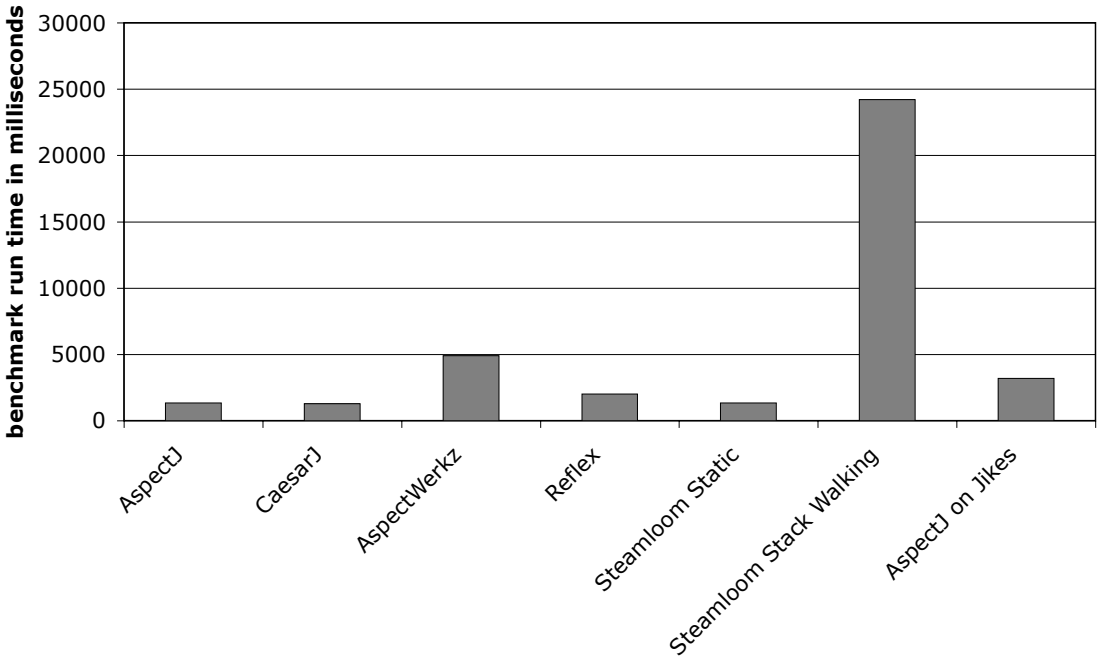

Figure 4.18.: Results for the nested control flow benchmark.

Summary

The above results clearly show dynamic pointcuts are most efficient when they are supported at VM level. Still, exploiting VM-internal structures does not necessarily yield better performance when the approach itself suffers from conceptual weaknesses. Stack walking has not proven to be a reasonable solution since its complexity depends on the stack depth met at join point shadows. Continuous weaving does not yield satisfactory performance unless control flows are *very* infrequently entered and left.

Counters, exhibiting constant cost at control flow entries and exits as well as dependent shadows, appear to be the best solution for matching control flows. When they are given dedicated support from the run-time environment, they even gain some more efficiency.

4.2.7. Weaving Performance

The performance of weaving is especially interesting when systems allowing for dynamic weaving are used. In such systems, an application may even noticeably stall when dynamic weaving consumes much time, so AOP implementations should strive to provide fast dynamic weaving capabilities.

Many systems with support for dynamic weaving perform preparation steps at load-time, which leads to a certain impact on class loading. In Sec. 4.2.2, some measurements pertaining to class loading have been made; they will be recapitulated below. However, the measurements made there have not explicitly taken into account the cost imposed on class loading for actually *weaving* aspects into application code, or for *preparing* join

point shadows for later dynamic weaving.

The measurements conducted in this section follow a straightforward approach. They measure the time it takes a certain AOP implementation to weave a certain aspect that affects a large number of join point shadows. Weaving is regarded as a two-step process: it consists of pointcut evaluation, and the decoration of join point shadows with advice invocations and residual code.

Pointcut evaluation calls for a differentiated measurement approach because of the varying complexity of join point shadow retrieval for different kinds of pointcuts. Resolving a `execution` pointcut is mostly trivial because it basically consists of finding all methods that match a given name pattern. Conversely, resolving pointcuts such as `call`, `get` or `set` is far more complex because the appertaining join point shadows, e. g., call sites of methods, may be spread all over the application in question.

To gather details about weaving performance in the regarded AOP implementations, the following measurements are conducted:

- *Class loading:* the time to load a large number of classes is measured for the case when a certain number of join point shadows is contained.
- *Dynamic deployment:* the time to deploy an aspect affecting a large number of join points is measured.

In both cases, an equal number of all kinds of pointcuts is used.

For measuring dynamic weaving performance, two times at which weaving may occur must be taken into account: class loading and actual dynamic weaving at runtime. Therefore, the following approach to measuring dynamic weaving performance is adopted.

A group of 20 dummy classes is used. Each of them has 20 methods:

- 5 `private void` methods,
- 5 `public void` methods that forward to the private ones,
- 5 `private Object` methods, and
- 5 `public Object` methods that forward to the private ones.

Moreover, each class has a default constructor.

The classes are organised in pairs `(X,Y)` where `X` has a member of type `Y`, and the private methods in `X` forward to the corresponding public methods in `Y`. That way, a certain degree of intra- and inter-class coupling is achieved which is interesting for the measurements.

An aspect is used that attaches a simple before advice to all calls of `public void` methods and to all executions of `public Object` methods in the dummy classes. In case a system does not support method call join points, execution join points of the respective methods are decorated instead. All in all, 150 join point shadows have to be retrieved and instrumented (50 call and 100 execution shadows).

The measurement itself works as follows:

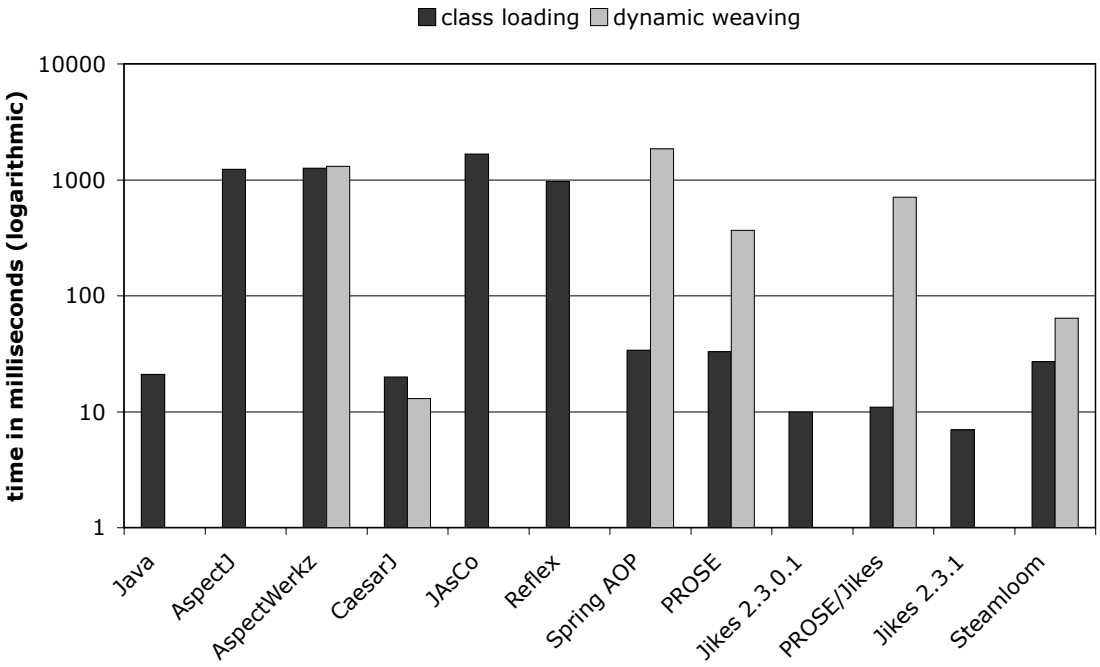

Figure 4.19.: Dynamic weaving performance measurement results.

- The measurement application starts up and explicitly loads the dummy classes via `Class.forName()`. The time taken for this is measured.

- If the system being measured supports actual dynamic weaving (e. g., AspectJ does *not*), the aforementioned aspect is dynamically deployed, and the time this takes is also measured.

AspectJ was taken into account because of its support for load-time weaving, which is interesting to compare with dynamic weaving approaches.

Some restrictions apply. As mentioned above, some systems do not support method call join points, in which cases execution join points were used instead. These systems are JAsCo and Spring AOP. Moreover, AspectJ and JAsCo do not support actual run-time weaving: for them, only load-time weaving cost was measured.

The results from these measurements are gathered in Fig. 4.19. Apart from results for the measured AOP systems, the underlying execution environments' performances for class loading are also shown to give a better impression of the actual cost. All results are given in milliseconds.

The standard JVM-based systems performing either preparatory steps (AspectWerkz, Reflex) or the entire weaving process (AspectJ, JAsCo) at load-time all exhibit very high costs for class loading. AspectJ's load-time weaver exhibits the same cost as that of AspectWerkz, but the latter moreover has an almost as high cost when the aspect is actually deployed at run-time.

Spring AOP does not make class loading itself significantly more expensive, but dynamic proxy creation is costly. The two PROSE implementations also are moderately more expensive during class loading. Their dynamic weaving process is, however, expensive.

CaesarJ weaves all aspects into the application at compile-time and performs no additional operations at load-time. Dynamic deployment is also very fast, because it merely consists of adding entries to internal data structures.

Reflex *does* support dynamic weaving by means of (de)activating behavioural links. The reason for no value being shown for Reflex in the figure is due to the fact that it implements dynamic (de)activation by simply setting a flag in an activation condition, which operation is too fast to have a footprint.

Steamloom performs very well at dynamic weaving. Note that the dynamic weaving step contains, in the case of Steamloom, pointcut evaluation *and* bytecode instrumentation. This shows that building and maintaining indices for, among others, method call join points is beneficial. The impact on class loading, effected by building indices, has already been observed and discussed in Sec. 4.2.2.

4.2.8. Memory Consumption

The amount of memory used by an application has a direct influence on the frequency of garbage collections, which in turn reflect on the overall performance of the running application. An application that normally exhibits no frugal memory consumption should do so when run in an AOP environment as well, i. e., the AOP environment should be implemented in a way that it avoids excessive memory requirements.

To evaluate the regarded AOP implementations' memory consumption behaviour, two kinds of measurements were conducted, namely overhead measurements and measurements addressing AOP functionality.

The overhead measurements were conducted using the DaCapo benchmark suite [45]. It consists of a number of real-world Java applications that have high memory usage. Since the garbage collector affects the overall execution time, results for DaCapo benchmarks are expressed in the time that a JVM needs to execute them.

For measuring the memory impact of AOP functionality, the *dynamic weaving* measurements from Sec. 4.2.7 are reused. To obtain utilisable results, the respective run-time environments were instructed to log garbage collection behaviour and output memory statistics.

DaCapo Benchmarks

Only a selection of the DaCapo benchmark applications was run, because some of them are not executable on the Jikes RVM. The applications used for generating the results presented below were these:

- the *ANTLR* parser generator,
- *Bloat*, which analyses and optimises Java bytecode,

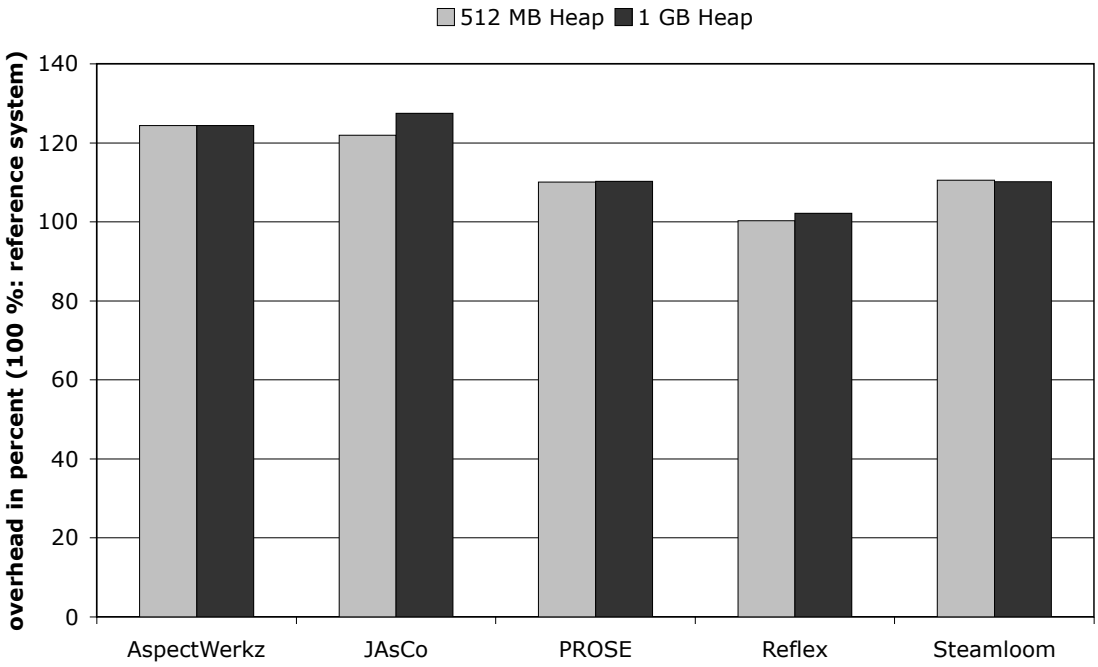

Figure 4.20.: Results from DaCapo memory benchmark runs.

- an *HSQLDB* application, executing a number of transactions on an in-memory database,
- a number of Python programs, interpreted by *Jython*, and
- a series of XML document transformations to HTML using *Xalan*.

Each of the DaCapo benchmarks can be run at three sizes: small, default, and large, where the latter is intended for generating actual results.

These benchmarks were again run on those systems that apply modifications to the class loading process or run-time environment. As with the SPECjbb2000 benchmark in Sec. 4.2.2 above, two heap sizes, namely 512 MB and 1 GB, were used. The benchmark size was set to *default*, because both versions of the Jikes RVM were apparently not able to execute a *large* benchmark and crashed during its execution. This appears to be due to a problem inside the Jikes RVM *itself*. PROSE/Jikes crashed reproducibly even during the reduced measurements, for which reason results for this system are not provided.

The results are gathered in Fig. 4.20. They represent average overheads computed over all of the benchmark applications that were used.

The systems that already exhibited a high class loading overhead also exhibit the highest overheads in the DaCapo benchmarks. AspectWerkz and JAsCo, as mentioned in Sec. 4.2.2, perform comparatively extensive checks and even transformations of bytecode to an internal representation even if no aspects are present. It is obvious that such an approach has some impact on memory usage.

Reflex benefits from its efficient class loading approach also in terms of memory consumption. No significant overhead was observed.

The overheads ascertained for PROSE and Steamloom are moderate. For Steamloom, it is obviously the amount of memory required to represent method bytecodes as linked lists of instruction objects that carries weight. When executing PROSE, the VM, running in debug mode, cannot apply all optimisations it could apply otherwise, which results in an overhead.

The heap size does not significantly matter. This is because the memory loads that the DaCapo benchmarks create are not as aggressive as that of SPECjbb2000. The latter leads to much more frequent heap exhaustion for smaller heaps.

Dynamic Weaving

When regarding the amount of memory consumed by an AOP implementation during dynamic weaving, the following three characteristics are interesting:

- the basic memory *overhead*, due to internal data structures of the AOP implementation,
- the amount of memory allocated *during class loading*, because a class loader that weaves aspects or prepares classes for later weaving may allocate objects for the internal representation of classes (e. g., as `byte` arrays), and
- the amount of memory allocated *during actual dynamic weaving*.

The total memory allocated is also interesting, but merely as a summary overview.

For measuring memory consumption during dynamic weaving, the application used in Sec. 4.2.7 was modified to output the amount of free heap space at three points in time:

- prior to loading the dummy classes,
- after loading the dummy classes, and
- after dynamically weaving the aspect.

Moreover, all VMs were run with a fixed heap size of 512 MB, and explicit garbage collection was disabled. Since totally disabling garbage collection is not possible on the standard JVM, the VM was instructed to log garbage collection events.

The results obtained for the three characteristics described above are displayed in Fig. 4.21. In addition, the total amount of allocated memory is given in Tab. 4.2. All numbers are also given for the employed base run-time environments, and the table contains overheads of the various AOP implementations over their respective base environment. The overheads are not given in percent, but express how many times as much memory an AOP implementation allocates when compared to the base environment.

There are three possible reasons for the observed environment overheads. The first is that a system employs a JVMTI agent that is initialised prior to the execution of the application's `main()` method and hence has already allocated memory before the

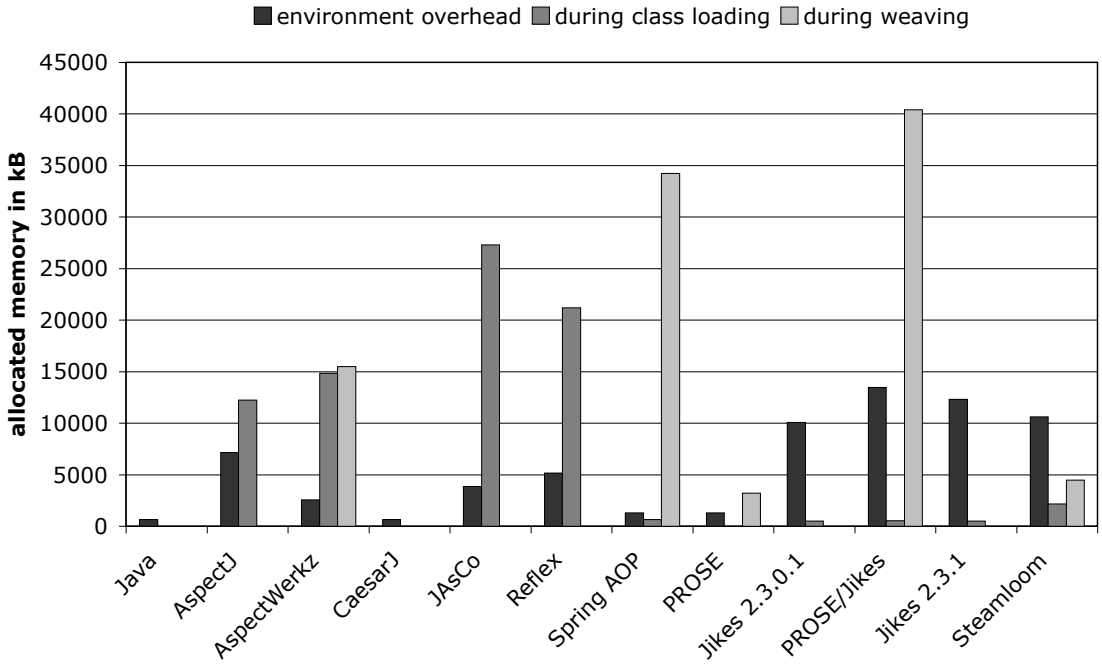

Figure 4.21.: Memory allocation characteristics due to dynamic weaving.

application actually starts. This holds for AspectJ, AspectWerkz and JAsCo. AspectJ allocates the most memory prior to executing the application due to its weaver's memory requirements.

The second reason is that an application is executed in a harness. This applies for Reflex, which starts the actual application from within its own program runner. The Reflex runner installs a dedicated class loader that is initialised before the application is run.

For the systems based on the Jikes RVM, the third reason applies: it is the shared heap where VM-internal and application objects live together. Jikes' internal objects amount to some 10–12 MB (this can be seen from the results for Jikes 2.3.0.1 and 2.3.1). Regarding this, the overhead imposed by PROSE/Jikes is reasonable and due to internal data structures of the modified run-time environment.

Steamloom even has a slightly *smaller* footprint than Jikes 2.3.1. This is, though it may sound contradictory, due to its larger boot image. Steamloom's boot image in fact contains more classes than that of Jikes, and some of these classes are from the standard libraries. These classes need not be loaded because they are already present. Hence, no objects that would otherwise be needed during loading—e. g., streams for reading class files, or other objects for the intermediate representation of entities being loaded—have to be allocated. Since the unmodified Jikes RVM loads a certain number of classes from the standard libraries prior to the execution of the measurement application's `main()` method, it allocates the objects needed during loading. Steamloom comes upon them in the boot image—they are not loaded, and no objects for class loading are created.

	total allocated	**overhead**
Java	645.2 kB	N. A.
AspectJ	19,400.0 kB	30.1
AspectWerkz	32,911.5 kB	51.0
CaesarJ	645.2 kB	1.0
JAsCo	31,188.9 kB	48.3
Reflex	26,385.6 kB	40.9
Spring AOP	36,159.6 kB	56.0
PROSE	4,520.1 kB	7.0
Jikes 2.3.0.1	10,600.0 kB	N. A.
PROSE/Jikes	54,424.0 kB	5.1
Jikes 2.3.1	12,824.0 kB	N. A.
Steamloom	17,260.0 kB	1.3

Table 4.2.: Total memory allocated in run-time environments.

The approaches that do not employ agents (CaesarJ, Spring AOP, and PROSE) have no or very little overheads. Spring AOP's slight overhead is due to the Spring AOP measurement application allocating more objects than the measurement application for the other approaches. Due to Spring's different programming model, it was necessary to provide a dedicated implementation of the measurements. PROSE, running in debug mode, slightly suffers from this.

During class loading, large amounts of memory are allocated in those systems that have a weaving class loader, which was expected. Spring AOP, due to its coupling to parts of the Spring framework, needs to have a comparatively big number of JAR files in the class path, which results in increased allocation behaviour. The Jikes-based systems all exhibit some allocation because objects representing the loaded classes and methods are created on the shared heap. Steamloom allocates significantly more memory because it builds BAT data structures.

Dynamic weaving exhibits interesting allocation characteristics of the various systems. AspectWerkz again needs to create considerable amounts of ASM objects to parse and modify method bytecodes. Spring AOP dynamically creates proxy classes and objects.

CaesarJ and Reflex exhibit no allocation. In CaesarJ, no allocation takes place, but internal data structures are updated. In Reflex, dynamic deployment consists of simply setting a flag.

PROSE actually retrieves join point shadows during the weaving step, which process comprises the creation of numerous internal objects and data structures. PROSE/Jikes, additionally, *clones* methods for weaving, which induces a high allocation rate.

Steamloom, on the other hand, does not clone methods but modifies them in place and does therefore not allocate that much memory. It also creates filter objects for join point shadow retrieval.

Regarding the total allocation data gathered in Tab. 4.2, it can be concluded that, of the AOP implementations capable of dynamic weaving, CaesarJ and the debugger-

based version of PROSE are the most economical. However, they exhibit weak run-time performance when advice are attached to join points (cf. Sec. 4.2.3).

Of the systems with high performance, Steamloom is the one allocating the least amounts of memory. Its characteristics must also be seen in conjunction with the fact that VM-internal and application objects share the same heap. Regarding the overhead, Steamloom exhibits the best memory characteristics of the fully dynamic systems.

Summary

From the measurement results presented in this section, it can be concluded that load-time weaving approaches and proxy-based implementations have the worst memory characteristics. The fully dynamic systems relying on actual services of the underlying execution environment or even extending it, namely Steamloom and the two versions of PROSE, have the smallest overheads in allocation.

The shared heap and boot image problems make it hard to argue with accurate memory allocation overheads in the cases of PROSE/Jikes and Steamloom. Nevertheless, the allocation characteristics met in the respective unmodified Jikes RVMs can be regarded as a "base environment allocation" that occurs anyway. This amount can be subtracted from that measured for the two AOP implementations to yield an amount of allocation that happens "due to AOP". When this is done PROSE/Jikes allocates 42.8 MB, and Steamloom allocates 4.3 MB.

These numbers move Steamloom very close to the standard JVM-based PROSE, which otherwise exhibits the best allocation behaviour of all fully dynamic systems. Thus, it can be concluded that relying on services of the underlying execution environment or extending it is the best option for implementing dynamic AOP with respect to memory consumption.

The great benefit of Steamloom in this respect is that it does not have to allocate large amounts of memory dedicated to the representation of otherwise VM-internal structures, i. e., method bytecodes. Steamloom operates on the VM-internal representation of methods. Other approaches that modify method bytecodes have to create dedicated objects, manipulate them, and pass them to the virtual machine for reinstallation.

4.3. Technical Criteria Assessment

This section deals with technical implementation details one level above raw performance data. More precisely, the issues addressed here deal with the integration of AOP implementations with the execution environment of the base language used.

The focus in the following assessments is on the traces that an AOP implementation leaves in a base application's code or in the memory of the base language's execution environment. Considerations include, on the one hand, the effort put into manipulating the application's representation for weaving. On the other hand, the shape of woven code and its impact on the complexity of executing the application are analysed.

4.3.1. Representational Overhead

An AOP implementation needs to have some kind of access to some representation of the base application mainly for two purposes:

- During pointcut evaluation, the AOP system must be able to find join point shadows in base application code.
- Weaving frequently requires modifying the application's code, which must be made available for modification.

Pointcut evaluation is, in PA-flavoured AOP language implementations on the Java platform, normally done based on the bytecode representation of the application. This is also often used to weave aspect code into the base application.

For pointcut evaluation, sometimes the bytecode representation is not semantically rich enough to satisfy the needs imposed by the expressiveness of the pointcut language. In such cases, other approaches to obtaining the required information from the application are possible as well. For example, the object heap can be made accessible to analyse object interdependencies at run-time [123].

Unless an AOP implementation is able to access the base language execution environment's *internal* representations of application models, making them available usually comes at a certain price. A redundant representation of application elements certainly requires additional memory. But it also impacts performance, because the extraction of method bytecodes from the VM for modification and their reinstallation into the VM after weaving both have some cost.

It is the purpose of this section to analyse the efforts that different AOP systems make to address the two requirements mentioned above, and to classify them into three groups:

- An AOP implementation induces a *low* representational overhead if it is able to work on an irredundant representation of the exploited application models.
- Where a redundant representation of the application model is used, the overhead is considered *high*.

A redundant representation of application models mainly has an impact on memory consumption, because environment-internal data are duplicated at application level. Moreover, the redundantly represented model entities must be submitted to the execution layer to be actually used, which is an additional indirection that may impact the overall performance. Hence, a low representational overhead is generally desirable. Ideally, an AOP implementation directly manipulates the base environment's internal models themselves.

The analysis conducted in this section is based on the descriptions of the AOP systems in Ch. 2 and on the measurement results anent weaving performance and memory consumption in Secs. 4.2.7 and 4.2.8.

Systems that can be classified to have a *high* representational overhead are AspectJ (when load-time weaving is used), AspectWerkz, JAsCo, both versions of PROSE, Reflex, and Spring AOP.

AspectJ, JAsCo and Reflex extract class bytecodes and lift them to the representation used internally by a bytecode toolkit (BCEL or Javassist) they employ to find join point shadows and for weaving. After that, modified classes are passed on to the class loader as raw `byte` arrays.

AspectWerkz also uses a bytecode toolkit (ASM) and also modifies classes at load-time. Moreover, it creates `JoinPoint` classes dynamically as they are needed. For each generation of such a class, an ASM representation of the corresponding bytecodes is built and manipulated, which is then passed to the class loader in the form of a `byte` array, as above.

PROSE/Jikes operates on the virtual machine's internal representation of Java bytecodes, but it lifts them to a BCEL representation (like the AspectJ load-time weaver) for manipulation before they are reinstalled. PROSE running on the standard JVM relies on debugger breakpoints to signal join points at run-time, but it also creates a BCEL representation of loaded classes to identify the places in code where it has to register breakpoints.

In the case of Spring AOP, dynamic proxy generation is done by directly writing bytecode instructions to a stream to constitute a valid class file which is then passed to the class loader. This is also a redundant representation, like with other AOP implementations that operate at load-time. The only difference is that Spring AOP does not first transform bytecodes of a preexisting class to some internal representation.

The other systems, namely Arachne, AspectJ (when load-time weaving is *not* used), AspectS, CaesarJ, and Steamloom, have a *low* representational overhead. For the purely compiler-based approaches AspectJ and CaesarJ, this is immediately apparent, since they do not manipulate or query the application models at run-time.

Arachne, AspectS, and Steamloom *directly* operate on the internal representation of the application that is used by the run-time environment. In Arachne's case, this is simply the native machine code of the running application. AspectS operates on the meta-model of the application, which is available anyway in a Smalltalk environment. Steamloom performs bytecode manipulation like many of the above tools, but the representation of bytecode is not deliberately extracted from the VM, manipulated, and reinstalled.

Setting these considerations in relation to the measurement results gathered for dynamic weaving and memory consumption, it becomes evident that those systems exhibiting high memory allocation overheads according to Tab. 4.2 (p. 193) also have a high representational overhead. These systems also take the longest times for dynamic weaving and load-time preparation (see Fig. 4.19, p. 188).

From these observations, it can be inferred that a high representational overhead—expressed in the employment of bytecode toolkits for the representation, manipulation, and possibly later reinstallation into the VM—negatively influences both memory allocation behaviour and performance. The example of Steamloom shows that integrating AOP functionality with the VM allows for them both operating on the same represen-

tation of the application, leading to a more moderate memory consumption and better dynamic weaving performance.

4.3.2. Infrastructural Code

The amount of infrastructural code that has to be executed as part of the application can be expressed in three categories: low, medium, and high. They are characterised as follows:

- The amount of infrastructure is considered *low* if advice and residues are invoked implicitly by the run-time environment, without the need for the latter to invoke any specific application infrastructural code at application level. Simple retrieval operations, e. g., for advice instances, that do not comprise querying complicated data structures also fall in this category.

- *Medium* applies when advice and residues are invoked directly, but when a small amount of control flow in the application needs to be executed by the VM. An amount of infrastructure is considered "small" when no complicated control structures are involved, or when indirections are introduced into base application code.

- Infrastructure presence is *high* if the execution environment must, in order to invoke advice or evaluate residues, execute a significant amount of code that does not originally belong to the application, but to the AOP system. This also includes the maintenance of complex data structures.

The assessment for this does not address performance because its purpose not to discuss performance, which has been done at length in Sec. 4.2. It is rather the amount of code that is executed by the execution environment *in application space* that is of interest here. Hence, the analysis in this section focuses on discovering the complexity of the infrastructures involved when executing residues and advice in the various regarded AOP implementations. This is addressed by deriving an assignment of the AOP implementations presented in Ch. 2 to the above categories from the systems' descriptions.

It is also important to take into account infrastructure that is immediately perceived by end users when they are debugging an application running in an AOP-enabling environment. When an aspect-oriented application is debugged, infrastructural code can be distracting because it hinders a developer from instantly seeing the relations between join point shadows in the base application and advice executions attached to them. This is due to infrastructural methods turning up in the call stack, possibly consuming several entries. A developer *using* an AOP environment typically does not want to debug the AOP environment, but the application that is built using it. It is certainly possible to implement a debugger in such a way that is suppresses the output of infrastructural code, but this only shifts the effort to the implementor of the debugger.

The assessment of debugging is conducted by examining stack traces. The examination yields information about the amount of infrastructural code pertaining to the AOP system that becomes visible when, e. g, an exception is thrown in advice code and shall be traced back to the point in the base application where the advice call originated.

	advice invocations	residues
AspectJ	low	high
CaesarJ	high	high
Arachne	N/A	N/A
JAsCo	medium	high
AspectWerkz	medium	high
PROSE	high	high
Spring AOP	high	high
Reflex	high	high
AspectS	N/A	N/A
Steamloom	low	low

Table 4.3.: Amount of infrastructural code executed in AOP implementations.

Infrastructure Discussion

The results of assigning the systems presented in Ch. 2 to the categories introduced above are shown in Tab. 4.3. Two categorisations have been made based on the presentations of the systems in Ch. 2: one with regard to the amount of infrastructure involved when advice are simply invoked, and one with regard to the amount of infrastructure when `cflow` residues are involved. Only one line is given for both AspectJ and PROSE, since the two versions of each of them that were taken into account during the measurements in Sec. 4.2 do not differ with regard to the amount of infrastructure they involve.

Two systems need to be dealt with *a priori* and are excluded from the ensuing discussion: Arachne and AspectS cannot be assigned to any of the categories unambiguously. For both systems, this is because they basically draw no distinction between run-time environment and application so that it cannot be clearly determined what portion of functionality is actually executed in application space and in the run-time layer.

In Arachne, all code is native machine code throughout, no code is represented at a higher level of abstraction and then executed by some entity like a virtual machine. Hence, it could be argued that Arachne exhibits a *high* overhead in all cases because it executes all code in the application space. On the other hand, the way it realises advice invocations is minimalistic, calling for it being classified as having a *low* infrastructural impact.

In AspectS, all code is executed by the virtual machine, including the interpretation of functionality constituting the language execution model itself. The way AspectS realises AOP functionality could be interpreted in a way that allows for it being assigned to the *low* category since meta-level entities are manipulated. Still, since meta-level entities are also part of the application, the *high* category would seem appropriate as well.

For advice invocations, systems with high infrastructure amount involved in advice invocations are CaesarJ, PROSE, Spring AOP, and Reflex. For PROSE and Spring AOP, the respective sequence diagrams (cf. pp. 69, 76) clearly make this evident. CaesarJ invokes advice via an indirection and also executes some amount of conditional logic in

the control flow leading to advice invocations. Reflex also involves explicit activation checks at each join point shadow.

Medium infrastructure presence is stated for JAsCo and AspectWerkz because they introduce indirections, but do not execute the amount of conditional logic that is met in Reflex and CaesarJ, for example.

AspectJ and Steamloom have a low infrastructure presence for plain advice invocations. AspectJ just performs a static method call to retrieve the aspect instance and then immediately invokes the advice. Unless more complex constructs, such as `perthis`, are used, the lookup of the aspect instance is extremely cheap. Steamloom only performs the actual advice invocation at application level; advice instance lookup is implemented in the virtual machine.

For `cflow` residue executions, all systems but Steamloom have been classified as having a high amount of infrastructure being executed at application level. For CaesarJ, PROSE, Spring AOP, and Reflex, this is immediately apparent since they have already a high amount of infrastructure involved in plain advice invocations.

JAsCo uses stack walking to match `cflow`s, which involves a considerable amount of code to iterate over the single entries of a stack trace, and additional conditional logic to match the methods against the pointcut constituting the control flow. AspectWerkz does not use counters, but maintains a stack in any case.

AspectJ relies on `ThreadLocal` objects to maintain control flow counters. While this approach is conceptually close to that adopted in Steamloom—`ThreadLocal`s are kept in `Thread` instances for implicit mapping—, it still involves large amounts of code being executed at application level.

Steamloom inserts invocations to static residual methods into the application bytecode at control flow entries/exits and at dependent shadows. These methods are *not* executed as part of the application; they are direct entry points into the `cflow` management logic provided by the virtual machine. All data structures associated with control flow management are also kept in the VM itself. Hence, Steamloom was assigned to the category of systems with a *low* amount of infrastructure.

Perception of Infrastructure during Debugging

The analysis in this section is based on a simple experiment that was run on all systems. An application similar to the sample aspect from Ch. 2 was used. It invokes a method, and a before advice is attached to the call. Both the advice and decorated method print their respective stack traces. To that end, they create a `Throwable` instance and invoke `printStackTrace()` on it.

From the stack traces, the number of method invocations between the call site and the actual call target was counted. A value of 1 indicates that a method is called immediately, a value of 2 means that one infrastructural method lies in between the call site and the original target method.

The results can be seen in Fig. 4.22. It must be noted that, in case of JAsCo, PROSE, and Spring AOP, method *executions* were advised rather than method calls, because these systems do not support the latter. Also, AspectJ and PROSE occur only once.

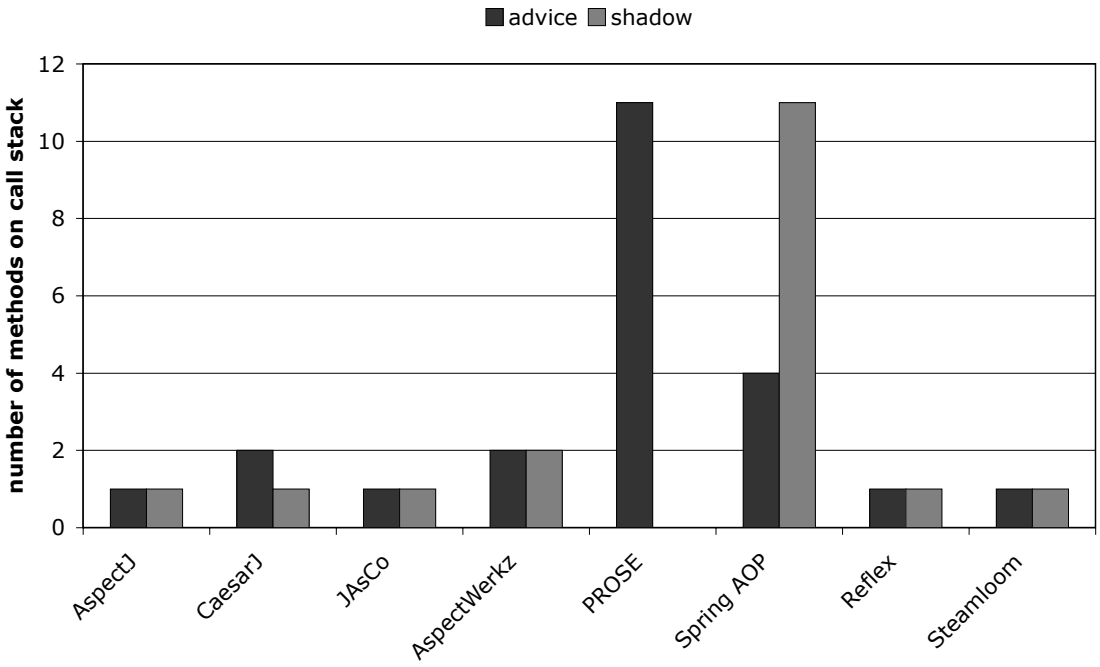

Figure 4.22.: Number of methods on the call stack when methods are advised.

This is because AspectJ 5 and AspectJ 1.2 as well as PROSE and PROSE/Jikes yielded the same results for this experiment.

The results correlate with the sequence diagrams presented in Ch. 2, and with the above preliminary classification. Those systems for which complex sequence diagrams were presented also have the highest amount of infrastructural code on the stack when they invoke advice or original shadows.

For PROSE, only a value is given for the number of methods that are on the stack when the advice is invoked. PROSE supports method execution join points, not calls, and all that is woven at a method entry is an invocation of a callback method in the PROSE infrastructure, after which execution proceeds normally.

JAsCo and Spring AOP also only support method execution join points. Still, they both exhibit infrastructure on the call stack. JAsCo does so because it *replaces* the original method body with a new one that invokes the advice. So, an additional call becomes necessary to invoke the method to whose execution the advice is attached.

In the case of Spring AOP, the invocation of the decorated method involves significantly more infrastructure than the invocation of the advice. This is because the decorated method is actually invoked from within the infrastructure, which can be seen from the sequence diagram presented for Spring AOP (Fig. 2.8, p. 76).

Reflex has a relatively complicated control flow to invoke an advice (cf. Fig. 2.10, p. 84), but advice and decorated method are *directly* invoked.

Summary

The above considerations have been derived without regarding performance measurement results. The conclusion that a high amount of infrastructure always entails weak performance is not justified; systems exhibiting a high amount of infrastructure, especially when `cflow` residues are to be executed, still may exhibit very good performance. Examples of such systems are AspectJ, CaesarJ and Reflex. Even if infrastructure is significant in terms of code being executed, it can still be implemented in a very efficient way.

The question of infrastructure is of more interest when the complexity of a system is regarded, e. g., for maintenance and comprehension. Shifting infrastructural code away from the application level and moving it into the execution layer enhances comprehensibility of woven code, because the code can immediately express *what* it does instead of *how* it achieves its goals. Low infrastructure presence in code, e. g., because a dedicated bytecode like Steamloom's `aaitpush` is used instead of possibly complex advice instance retrieval operations, increases its understandability.

4.3.3. Java Security

This section focuses on the ways the different AOP implementations affect the security model of the Java programming language. The various security APIs that Java provides are *not* in the focus of this investigation. Rather, the discussion focuses on the low-level security model of the *core language* as provided by the virtual machine.

Low-level security is guaranteed by the following mechanisms and concepts [156, 121]:

1. *Well-defined and open language:* the Java programming language's implementation, i. e., the virtual machine, is an open standard [109]. Its implementation, the standard virtual machine, is available in source code.

2. *Static typing and compile-time type checking:* the static type system of the Java programming language is enforced to a great degree already at compile-time, which makes the executed code type-safe. The compiler also enforces that all variables must be initialised prior to the first access to them.

3. *No pointers and automatic memory management:* Java's lack of pointer arithmetic and, hence, its lack of possibilities to directly access memory, avoids multiple sources of error that are common in other languages, such as C. The fact that garbage collection as an inherent part of the execution platform attenuates the lack of direct memory manipulation in that a reliable and mostly transparent mechanism for memory management is provided.

4. *Class file verification:* class files are verified as they are loaded into the virtual machine. The verifier performs a wide range of analyses on the bytecodes contained in a class file. It rejects any class file that does not conform to the rules it defines.

The following discussion of the regarded AOP implementations takes into account the aforementioned properties of low-level security in Java and discusses in how far the

approaches guarantee them or are prone to breaking them. In the latter case, the risks coming with a particular approach are outlined.

Language Definition and Static Typing

It is the task of a language's compiler to ensure that programs adhere to the language definition, and—given that the language has a static type system—that they are soundly typed. Hence, these two points are of main interest for AOP implementations that are provided as language extensions, i. e., for those systems that bring their own compilers. Representatives of these systems are, in the context of this work, AspectJ, CaesarJ, Arachne, and JAsCo. Apart from Arachne, all of them extend the object-oriented type system of the base language with aspect-oriented types.

With respect to aspect-oriented extensions for object-oriented programming languages, it can be argued that AOP "breaks" with some principles of OOP, such as encapsulation. This is true insofar as aspects can, in some systems, have access even to private members of classes they advise. A discussion of the question whether crosscutting modules should also crosscut private state and behaviour is beyond the scope of this work.

It can be observed that some of the approaches presented in Ch. 2 bring features that allow for avoiding the risk of breaking encapsulation. In AspectJ, aspects have to be explicitly marked as `privileged` to be granted access to private members. Spring AOP, JAsCo and AspectS have a restricted join point model that allows for advising method executions (message receptions in AspectS) *only*. Private members are protected.

CaesarJ, in which classes and aspects are blended, brings a very strong type system of its own. It ensures consistent and type-safe use of crosscutting to a high degree, even in the context of dynamic deployment.

Type safety cannot be guaranteed in all cases. For example, in approaches that work *after* the compilation phase—i. e., at load-time or even at run-time—, static type analysis can be applied only to a limited degree. In the context of dynamic class loading, not all the typing data that the compiler could access is available at once. Moreover, run-time AOP approaches often employ reflection to invoke advice, which also carries weight.

Fortunately, the Java programming language brings strong support for reflection in the form of extensive checks. They ensure that only those methods are invoked on an object that are implemented in the interface of its class. They also ensure encapsulation. A range of exception classes is available, instances of which are thrown when illegal attempts to access a member or invoke a method are made. AOP implementations using reflection draw on these features of the language to avoid risks.

Steamloom and the two implementations of PROSE are, to varying degrees, extensions to the virtual machine *itself*. The version of PROSE running in the standard JVM employs a plugin written in C that addresses the VM's low-level interfaces to register debugger breakpoints for join points and to signal join point occurrences to the AOP infrastructure. The use of C as a programming language is problematic with respect to type safety: all Java objects are represented by C structures of the type `jobject`; there is no actual object-oriented interface to them.

A certain degree of type safety can be assured by providing access to the low-level

C functionality through Java interfaces that enforce types for parameters and return values. In addition to that, the C plugin must handle all Java objects with extreme care and employ mechanisms that guarantee type safety.

The Jikes RVM-based PROSE implementation extends the VM by a mechanism to modify method bytecode instructions and selectively recompile methods. Steamloom extends the VM in various regards. Both implementations operate on low-level structures of the VM just like the C plugin mentioned above, but they benefit from the Jikes RVM itself being implemented in Java, which guarantees type safety to some degree. As long as no direct manipulations on memory structures are done, type safety is enforced by the language.

Memory Management

Java's automatic memory management capabilities can only be significantly impaired by AOP implementations that operate at the same level as the memory management logic, i. e., in the virtual machine. Thus, only PROSE and Steamloom are of possible interest with regard to this. All other approaches have no chance to derogate memory integrity because they operate at application level, where all memory-related operations—allocation, deallocation, and referencing—are controlled by the memory management subsystem of the virtual machine.

When an AOP implementation is implemented at virtual machine level, it is possible that memory is allocated bypassing the memory manager; e. g., if the implementation of the virtual machine does not enforce that memory be allocated *solely* through the interface of the memory management unit. This, alone, is not yet a risk, though. Normally, the Java object heap, the virtual machine's internal heap and memory available to VM plugins are segregated. So, as long as the AOP implementation does not exceed the available memory, memory management is not corrupted.

The AOP implementation must, however, take care about *all* interactions with Java objects [90]. Java objects that are accessed from a plugin must be registered with the VM and explicitly released when the plugin does not need them any more. Also, the creation of objects in native code is problematic: objects must explicitly be allocated, initialised, and registered with the VM. Keeping track of these interactions is, when an application is complex enough, non-trivial and prone to introduce errors.

This complexity applies, of the systems presented in Ch. 2, only to PROSE on the standard JVM. The two Jikes extensions, being themselves implemented in Java, perform all memory allocation through the VM's memory manager. Memory integrity is thus guaranteed in PROSE/Jikes and Steamloom.

Class File Verification

Class file verification is a crucial means of the VM to enforce the execution of code that does not violate the constraints of the programming model. As mentioned in the introduction, the verifier checks all kinds of integrity constraints concerning class files, to make sure they conform to the class file format specification as given in the JVM

Specification [109].

The focus of this work is on methods, so that the verification for methods' bytecode instructions are of most interest here. For method bytecodes, the verifier also checks that they are of integrity. Integrity, in this respect, addresses the correctness of the JVM's internal state during the execution of Java methods, comprising of the following elements [156]:

- no stack over- or underflows may occur,
- only valid values may be loaded from and stored to variables,
- the parameters to all bytecode instructions must be correct, and
- no illegal data conversion may be attempted.

It is obvious that all AOP approaches that manipulate bytecode may, in theory, lead to conflicts with the byte code verifier. Actually, most of the Java-based approaches employ bytecode manipulation. The compiler-based approaches generate whole class files and conform to the class file format specification in doing so. The approaches relying on load-time weaving modify class files as they are loaded into the VM and pass them on to the class loader afterwards. Because they just "hook into" the standard class loading mechanism, the verifier can still guarantee that no illegal code is actually loaded and executed. The same holds for approaches using HotSwap: the class redefinition mechanisms in the standard VM do not bypass the verifier. In a nutshell, all approaches based on the standard JVM cannot introduce illegal code.

The Jikes RVM-based implementations, PROSE and Steamloom, are a different case. This discussion is hampered by the fact that the version of the Jikes RVM that served as the foundation of these two implementations does not have a functioning verifier. Therefore, the discussion refrains to merely outlining possible risks and solutions.

With respect to verification, PROSE is not critical. Into base application code, its weaver inserts invocations of static callback methods in an infrastructure class. There exists one such callback for each join point type (i. e., method entries and exits, and field read and write operations), and the signature is fixed. The code inserted by the PROSE weaver merely assembles the parameters for the callback invocation from the context of the method and invokes the callback. This leaves the stack in consistent state: the callbacks consume their parameters and return `void`, so they have no effect on the method execution as a whole. A possible verifier would not object to the callback invocations inserted by PROSE.

Conversely, Steamloom not only applies direct advice calls instead of standardised callback invocations, but it also brings new bytecodes: `peek`, `aaitpush`, `beginadvice` and `endadvice`. Advice invocations and residual logic are treated in the same way as PROSE and all other bytecode-manipulating approaches deal with them: the weaver implementation takes care of them leaving the stack in a consistent state, accessing correctly indexed values, and so forth. The bytecodes are more interesting with regard to verification.

All of the newly introduced bytecodes are *completely* integrated with the virtual machine, i. e., all mechanisms that are applied to the standard bytecodes have been implemented for the Steamloom bytecodes as well. That is,

- both the baseline and optimising compiler know how to handle the bytecodes and can generate appropriate code for them, and
- GC map generation, which is done during baseline compilation, retrieves correct typing information for them.

The `beginadvice` and `endadvice` bytecodes are unproblematic. They exist for tagging purposes only and are treated like a `nop` bytecode by compilers and GC map generator.

For the `peek` bytecode, special action is taken. Its semantics somewhat contradict those of standard Java bytecodes; its result is polymorphic. For a given parameter `k`, it does not simply return the value contained in the `k`-th slot of the operand stack, but the `k`-th *value* found on the stack. That is, if that value happens to be a reference or any other value occupying one slot, the result of `peek` also occupies one slot. If the value is a `long` or `double`, the result occupies *two* slots.

Had `peek` been a standard bytecode available for generation by a Java compiler, there would have been two versions of it: `peek` and `peek2`, analogous to `dup` and `dup2`, for peeking a one-slot or two-slot value. Its sole availability at VM level in woven code has led to a more "convenient" semantics, whose implementation was supported by the BAT bytecode framework.

An instance of the `peek` bytecode instruction class in BAT always knows how many slots its result occupies. This knowledge is established when the woven code containing a `peek` instruction is generated. The compilers and GC map generators exploit it and can rely on its correctness, and can ultimately generate correct code.

The `aaitpush` bytecode is delicate in two respects. On the one hand, it accepts a parameter that is an index into an array, the access to which is deliberately *not* subject to a bounds check for performance reasons. On the other hand, the result of an `aaitpush` bytecode is some advice instance of a particular class, but the AIT is an array of `Object`s; no type check is performed, again for performance reasons.

The correctness of `aaitpush` applications is guaranteed by the Steamloom infrastructure that controls which instances are stored at which indices in AITs. Advice instances are stored in AIT slots at deployment time, and the respective slot indices are reserved by the corresponding `DeployedAspectUnit`s (cf. 3.8.1), from which the weaver, when generating an `aaitpush` instruction, retrieves its parameter. Several `DeployedAspectUnit`s that represent advice invocations sent to the same advice instance may share the AIT slot, but the slot is *not* available for storing a new advice instance until the last of the deployed aspect units occupying it has been undeployed.

Apart from the aforementioned possible risks, the introduction of new bytecodes of course is a possible source of illegal code, e. g., when malicious software employs such bytecodes in contexts where they are not applicable. In such cases, a verifier would certainly be helpful. Rules that apply to the Steamloom bytecodes and that would have to be implemented in a verifier are:

- Since all of these instructions are inserted during *dynamic* weaving only, none of them may be met in a class file as it is loaded into the virtual machine.
- `beginadvice` and `endadvice` instructions must occur in pairs with the same advice ID as parameter.
- Blocks enclosed by `beginadvice` and `endadvice` instructions may be nested, but must not overlap.
- A `peek` or `aaitpush` bytecode instruction must not appear outside a block enclosed by `beginadvice` and `endadvice` instructions.
- The parameter to a `peek` instruction must not be greater than the current stack depth at its bytecode index.

A possible future implementation of Steamloom on a version of Jikes that has a functioning bytecode verifier will take these into account.

4.4. Integrating AOP Support with the Underlying Execution Environment

This section serves as a summary for the two preceding sections. The point of interest that is discussed here is the actual *integration* with the base language execution environment that the regarded AOP implementations exhibit. The term "integration" addresses, in this context, the degree to which an AOP implementation makes use of the internal mechanisms of the base environment that are useful for its functionality.

The degree of integration can be expressed as follows:

- It is *low* if the entire AOP functionality is implemented as part of the application, in the form of Java classes that are entirely subject to execution by the base environment.
- The degree of integration is *medium* if the AOP implementation uses another interface to the base environment than Java bytecodes. That is, services provided by the environment—other than being able to execute Java applications—have been exploited to realise part of the AOP functionality, e. g., the debugger infrastructure or plug-in interfaces.
- Finally, the degree can be called *high* if the AOP implementation is itself *part* of the base environment, making the latter actually an AOP execution environment.

Tab. 4.4 shows the overall level of integration that have been identified for the various AOP implementations presented in Ch. 2.

Arachne and AspectS clearly exhibit a high level of integration with the underlying execution environment. For Arachne, this is true because its AOP functionality directly operates on native code. AspectS, as a Smalltalk system, directly accesses meta-level entities of the run-time environment.

	integration
AspectJ	low
CaesarJ	low
Arachne	high
JAsCo	low
AspectWerkz	low
PROSE	medium
Spring AOP	low
Reflex	low
AspectS	high
Steamloom	high

Table 4.4.: Integration of AOP systems with the underlying execution environment.

Most of the Java-based systems implement *none* of their functionality at the level of the virtual machine. Hence, they have been classified as having a *low* level of integration. PROSE and Steamloom are exceptions. PROSE, in both of its implementations, has a closer connection to the virtual machine than the other Java-based systems: either it contributes a C plugin accessing the low-level interface of the VM, or it extends the VM itself to allow for dynamic recompilation of methods. Steamloom, being the only of the systems that actually has multiple aspects of AOP functionality *implemented in the VM*, certainly has a high degree of integration.

Setting this in relation to the results of the performance measurements conducted in Sec. 4.2, it is not possible to draw the conclusion that a high integration alone brings good performance. For example, AspectJ, relying solely on Java bytecode, performs as well as Steamloom. When, however, the focus is on *dynamic weaving*, it can safely be concluded that an implementation that is integrated with the virtual machine is beneficial in many respects:

- The performance of applications not employing aspects is not dramatically affected. Deactivated aspects leave no footprint in terms of code or performance impacts in the application.
- Core AOP mechanisms, such as advice invocations and context access, can be implemented very efficiently. This also holds for advanced features, like scoped aspects.
- Dynamic pointcuts benefit from the support of residues at virtual machine level.
- Dynamic weaving is efficiently supported, and the memory consumption is moderate, even though a comparatively expensive approach using linked lists of instruction objects has been taken.

Other systems with support for dynamic weaving that are not integrated with the virtual machine perform significantly worse in some respects.

The discussions in Sec. 4.3 underpin these results, coming to the conclusion that the avoidance of redundant representation of application structures and a low degree of infrastructure being executed at application level are beneficial. The security model of the Java programming language is not broken by the way Steamloom is implemented.

5. Concluding Remarks and Future Work

This last chapter is dedicated to summarising the contributions of the work presented in the preceding chapters. Moreover, it gives an overview of the limitations of the achieved, summarises "lessons learned" and gives directions for future work.

5.1. Summary of Contributions

The *main contribution* of this work is the implementation of Steamloom. Steamloom is the first virtual machine for the Java programming language that offers native VM-level support for several core mechanisms of the aspect-oriented programming paradigm. In providing this implementation, the aspect-oriented programming paradigm is equipped with an initial version of what it deserves: an execution layer that is fit for supporting its basic mechanisms.

In particular, Steamloom's contributions are as follows.

Steamloom employs an integrated approach to bytecode management, letting the AOP infrastructure and other parts of the VM work on the very same representation of application bytecodes. Thus, the virtual machine is not only able to run Java applications represented in this form, but can also query the same representation for join point shadows and use it to weave aspect code into the base application. Pointcut evaluation is, on the one hand, supported by the bytecode representation, and, on the other, by the introduction of indices for the optimised retrieval of certain kinds of join point shadows. This altogether avoids the creation of a redundant representation using bytecode toolkits, an approach that is frequently adopted in other AOP implementations.

Dynamic weaving is a core feature of Steamloom. It is very naturally integrated with the virtual machine's optimisation subsystem to allow for excellent performance even in the context of dynamic weaving. Other dynamic weaving approaches frequently exhibit performance penalties due to a certain amount of infrastructure they execute at application level. This is avoided in Steamloom by relying on the VM's capabilities of compilation and optimised recompilation of application methods.

Various concerns of AOP implementations are, in Steamloom, directly supported by the virtual machine itself. Advice instances—i. e., those objects that represent crosscutting state and behaviour—are managed internally. Also, complex residues for dynamic pointcuts, namely `cflow`, are implemented through internal structures that allow for a very efficient evaluation of such residues. Steamloom also facilitates scoped aspects, most notably scoped to single instances, by means of integration with the VM's object model. These contributions avoid the use of extensive infrastructural code that must otherwise be executed by the virtual machine, instead of being an integral part of it.

Steamloom, as a whole, is a platform on top of which aspect-oriented programming languages can be implemented. The Steamloom API can be targeted by a compiler. Steamloom's pointcut model and weaver are extensible, so that the addition of new features is possible.

A *second contribution* of this work is the Steamloom-based implementation and experimental evaluation of three different approaches to implementing `cflow` residues. For the first time, three different approaches, one using counters to monitor control flows, one using stack walking, and one using continuous weaving, have been implemented and evaluated *on a single platform*, yielding actually comparable results.

The *third contribution* is an assessment of many different AOP implementations, which has not been achieved at that big a scale before. The assessment addresses numerous aspects of AOP implementations, ranging from performance of advice invocations in various contexts over memory consumption to dynamic weaving speed, and connects the results to implementation characteristics. As a result, the assessment underpins the claim made for this work, namely that AOP mechanisms need to be integrated with run-time environments to gain full flexibility and performance.

The *fourth and last contribution* of this work is a comprehensive presentation of various AOP implementations along the lines of a unified presentation framework. In the presentation, high-level architectural characteristics of the surveyed systems have been dealt with as well as implementation details of core AOP mechanisms. The framework served as the basis for a levelled comparison of AOP implementations and has made the aforementioned assessments more easy to devise.

5.2. Limitations of the Current Implementation

Steamloom, being the first implementation of VM-level AOP and a research prototype, is certainly not the last word on the subject. The implementation is limited in some respects. On the one hand, there are limitations in the range of AOP features that are supported by Steamloom. On the other hand, some technical limitations apply that make this particular implementation of VM-level AOP less efficient, in some cases, than such a VM ought to be. In the following paragraphs, the limitations will be briefly discussed, and proposals are formulated that outline possible implementation approaches to overcome them.

AOP Limitations With regard to AOP features, Steamloom is restricted in the following ways.

It has no support for *aspect instantiation granularity control*, as it is, e. g., provided through concepts like `perthis`, `pertarget`, etc., in AspectJ. Such support can be implemented by extending the AIT concept to contain different values for different objects (see also below).

It is not possible to *access the context of join points matched by a `cflow` pointcut*. To support this, stacks for monitoring such state have to be integrated with Steamloom. They can be supported in the same way as control flow counters are supported right now:

by being directly associated with internal thread representations and VM-level residues. It is also possible to extend the implementation in a way that allows context extraction directly from stack frames.

Aspect precedence declarations are not supported. Instead, aspect precedence is determined implicitly by the order in which they are dynamically deployed. While this solution delivers a clear semantics, explicit precedence declarations are certainly desirable. Their implementation mainly requires an augmentation of the weaver, so that it takes precedence relationships between aspects into account. Steamloom's weaving model, in which single *aspect units* are regarded as representatives of units of code that must be woven, allows for a fine-grained implementation of precedences.

Around advice do not exhibit optimal performance, and it is not possible to attach more than one around advice to one join point shadow. Moreover, around advice attached to method calls and executions that throw exceptions are not supported. This is due to the implementation of proceeding, which employs an assembler closure. These issues can be addressed by augmenting the optimising compiler to be able to generate the magic code involved with the invocation of the assembler closure, and by modifying the join point shadow retrieval logic to take proceeding points into account, which it currently does not do.

Instance-local scoped and normal aspects cannot safely be combined. The AIT concept is highly efficient, but since AITs exist, in principle, only *per class*, the creation of cloned TIBs with attached cloned AITs raises the level of complexity associated with AIT slot management dramatically. The introduction of dedicated AITs for *instance-local* decoration seems to be a solution to this problem. That way, every class would still maintain its own AIT, but each object that is decorated with instance-local advice would get an AIT of its own that does not interfere with the class-wide AIT.

Technical Limitations Moreover, there are some technical limitations.

Memory consumption is comparatively high because of the use of BAT instruction object lists as the internal bytecode representation. Moreover, join point shadow retrieval is still costly, even though indices are used. A solution to this problem, that facilitates join point shadow retrieval for *all* kinds of join point shadows in constant time, has been introduced [37].

In that approach, *all* join point shadows are replaced, e. g., at load-time, with invocations of so-called "envelope" methods that, per default, just execute the replaced shadow. For example, there is exactly *one* such envelope that is called for *each* writing access to a specific field. Envelope methods can be resolved in constant time, which brings a significant speedup for join point shadow retrieval. All weaving operations only modify the bytecode instructions of the envelope method.

Integration of envelope-based weaving with a virtual machine would serve two purposes: on the one hand, it would make the employment of a complex bytecode management toolkit unnecessary (and would thereby reduce the memory overhead), and it would increase the performance of join point shadow retrieval during dynamic weaving.

The *recompilation* approach followed by the dynamic weaving implementation entails

temporary slowdowns of the running application, until the optimising subsystem of the VM becomes aware of the deoptimised method. Moreover, cascading invalidation of compiled methods, which is necessary due to inlining, implies that more methods than actually necessary must be recompiled. This can be addressed by stronger relying on features that the optimising compiler brings, such as guarded inlining. In that case, no cascading invalidation would be needed, but all inlining locations of the originally (due to weaving) modified method could be invalidated by means of a guard.

5.3. Future Work

During the work on Steamloom, several points were identified that suggest directions for future work. This section discusses them and moreover reflects on next-generation aspect-oriented virtual machines.

Dynamic Weaving The employment of bytecode manipulation and ensuing recompilation of woven code is a convenient approach to implementing dynamic weaving at virtual machine level. Still, it inevitably introduces performance degradations regardless of what particular recompilation approach is adopted:

- Affected methods can be immediately recompiled, whereby all optimisations are retained: this stalls the application for some time, because, typically, an optimising compiler takes a comparatively long time to compile a method. Since dynamic weaving triggered by an explicit `deploy` statement should occur at once, recompiling such methods in threads running in the background is not acceptable, because weaving has to take effect immediately.

- Instead of an optimised recompilation, methods can be deoptimised, and the VM can fall back to their interpretation or recompilation at lower levels of optimisation: this introduces temporary slowdowns until the optimising subsystem of the VM has reoptimised the methods.

In a nutshell, bytecode manipulation and recompilation does not appear to be the ideal solution for high-performance support for AOP mechanisms. Future work in this respect will focus on devising mechanisms that allow to integrate such support even deeper with the execution model of the virtual machine.

To that end, weaving approaches adopted by AOP implementations such as Arachne can be taken into account. Arachne (cf. Sec. 2.4 weaves immediately on native code. It however suffers from the lack of expressiveness that native code normally has, and it strongly depends on symbol tables carrying rich information. Moreover, it is prohibitive with regard to inlining.

At virtual machine level, symbol tables and maps from native code to source code locations (i. e., join point shadows!) are available throughout because they are an inherent part of the execution model. Combining this rich information with the weaving approach of Arachne seems an interesting direction for research on more efficient dynamic weaving without the need for recompiling methods.

Control Flow Residues Providing efficient support for `cflow` is a highly interesting issue. Still, all approaches that appear to be efficient use counters to monitor control flows, which is conceptually unsatisfying as it is a very explicit way of expressing "this control flow is currently active".

Supporting conditions that check, for a particular join point, whether it occurs inside a given control flow, could be made more implicit and possibly efficient by exploiting the fact that method activations can be accessed as first-class entities at VM level. A method activation is normally represented as a stack frame. The stack frame carries, among others, information on the method whose execution has triggered its creation, on the stack frame from which the method was invoked, and so forth.

Stack walking has not proven to be an efficient solution to implementing `cflow` residues. However, stack walking is also a very explicit way of dealing with the check whether a stack frame is activated inside some control flow. Instead, the information whether a given activation constitutes a control flow could be stored in the activation *itself* (i. e., in the stack frame) and be propagated to other frames.

A solution attaching control flow information directly to stack frames would not only avoid both counter management and stack walking. It would, moreover, make thread-locality considerations completely superfluous, as stack frames are thread-local by nature. Such an approach would greatly simplify `cflow` residue management logic while likely improving performance.

Regarding this issue, future work will, in the first instance, focus on feasibility studies in languages that support activations as first-class entities, such as Smalltalk. An implementation in a future version of Steamloom will be provided once the implementation details will have been sorted out.

Optimised Index Structures In the present version of Steamloom, index data structures used to improve pointcut evaluation and to collect information on method inlining (cf. Secs. 3.8.2, 3.8.8) are implemented using collection classes from the Java API. While these exhibit good performance, their memory requirements are comparatively high.

On the one hand, it is thinkable to employ more optimised structures, such as bit sets, e. g., to represent inline locations, given that every method has a unique integral identifier. On the other hand, an alternative weaving approach, such as envelope-based weaving [37] (cf. above), would render indices for pointcut retrieval unnecessary.

Rich Pointcut Models Pointcuts are queries over the program execution. Program execution is, however, semantically much more rich than even `cflow` allows to express. For example, pointcut languages are being developed that take the relationships of objects on the heap into account to match pointcuts based on fine-grained state descriptions.

An example for an AOP language with a semantically richer pointcut language is Alpha [123]. It uses logic meta-programming in PROLOG syntax as its pointcut language to express pointcuts that can quantify over basically all aspects of program execution, including object relationships as well as past and future events. Another example are *stateful aspects*, which have been, of the systems presented in Ch. 2, implemented in

Arachne and JAsCo.

Such advanced concepts call for support from the execution layer. For example, the heap and object graph are immediately accessible from within the virtual machine. Exploiting the memory manager's knowledge about object interconnections would avoid the introduction of a redundant representation of the heap at application level.

Moreover, the execution history, which is especially important for stateful aspects, can be managed VM-internally by means of a modified call graph. The VM maintains a call graph anyway because it needs to monitor execution to decide on optimisations.

Static Crosscutting Static crosscutting, i. e., the introduction of new members, methods, super classes, etc. to classes to implement crosscutting, has not been discussed in this work so far. Steamloom, as an implementation, is explicitly intended to support AOP languages of the PA kind that solely focus on dynamic crosscutting.

Nevertheless, static crosscutting is of interest in dynamic weaving environments. For example, CaesarJ (cf. Sec. 2.3) has support for objects being "lifted" to other types as their original one, and to be addressed as if they were actual instances of these types. In CaesarJ, this is implemented using complex application-level data structures.

An implementation at virtual machine level could significantly improve this implementation with regard to performance. To support object lifting, the object model of the VM would have to be modified accordingly, to be able to represent an object's being an instance not only of a single type, but having several other "type facets" as well.

Benchmarking Judging about the large-scale performance of aspect-oriented applications is, currently, not possible for a wide range of AOP implementations. Some benchmarks have been provided for AspectJ [54], but they are not applicable to other AOP systems (cf. Sec. 4.2.

In devising the micro-measurement suite used for some of the performance evaluations in Sec. 4.2, some requirements for AOP benchmarks have been formulated [77]. One of the requirements is that an actual AOP benchmark should be *widely applicable*, another requirement demands *fairness*.

The aforementioned benchmarks, being applicable to AspectJ only, would have to be ported to all other systems to fulfil the first of the two requirements. Given that the different AOP implementations have gross differences in their programming models, the ported implementations would also look rather different. This raises the question of the second requirement: a benchmark should by all means not be designed for one AOP implementation and then be ported to others. Instead, it should be devised in abstract terms and then be implemented on all available platforms.

Future work in this regard will focus on devising a set of abstract benchmark descriptions, and on providing implementations for them.

Architectural Characteristics Steamloom has been implemented as an *extension* to an existing virtual machine, effectively *crosscutting* the underlying Jikes RVM. This

suggests to look at the implementation of Steamloom from the perspective of aspect-oriented programming.

Indeed, it is tempting to regard Steamloom as a large crosscutting concern and implement it using AOP techniques. However, the current join point models—as found in, e. g., AspectJ—are not fine-grained enough to express the close interrelations of the Jikes "base application" and the Steamloom "aspect" code. Nevertheless, possibilities to represent Steamloom as a clearly encapsulated crosscutting concern will be investigated.

A. Micro-Measurement Results

As announced in Sec. 4.2.3, this appendix gathers the micro-measurement results collected for all systems in their complete form. On the following pages, one figure is provided per system.

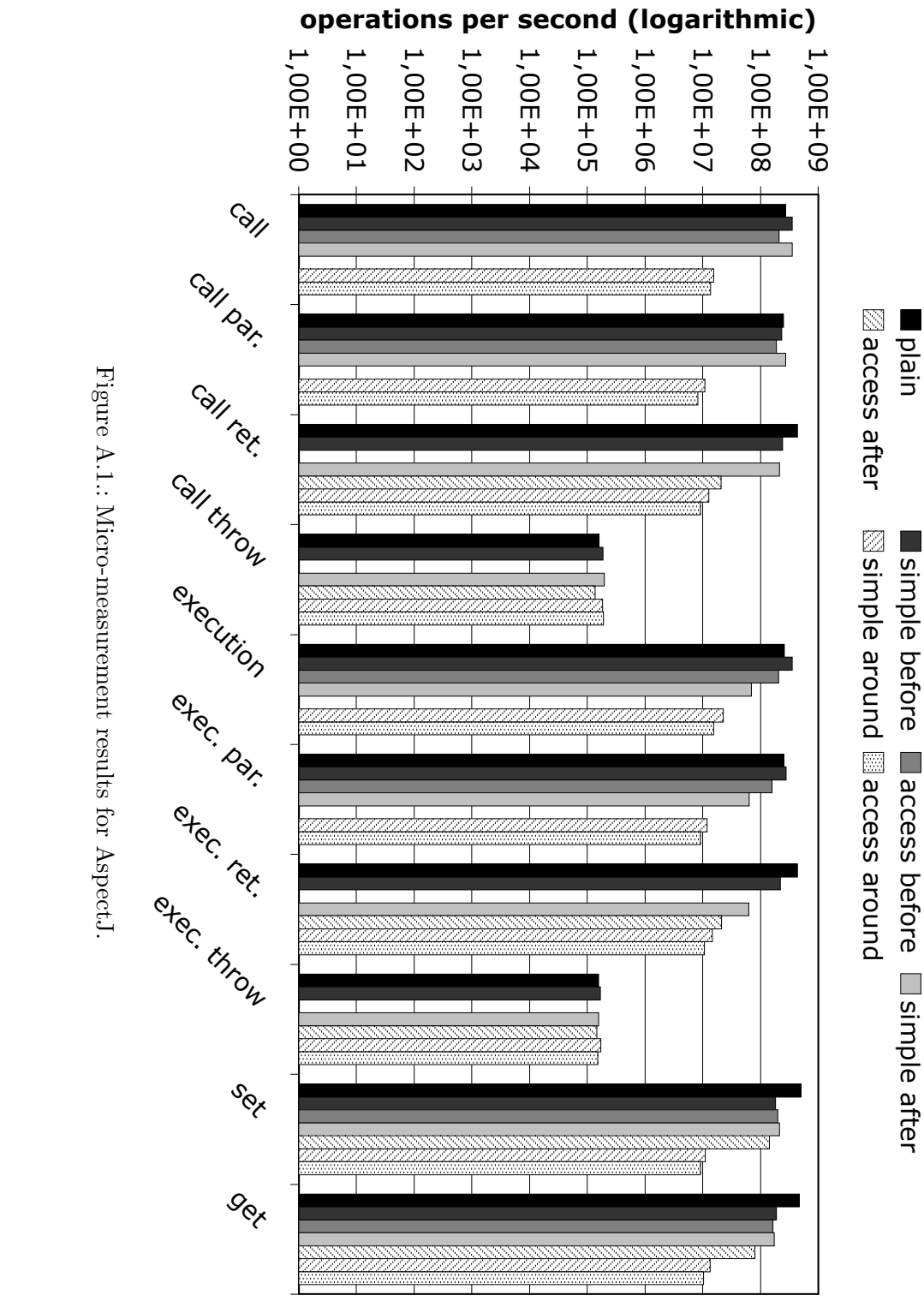

Figure A.1.: Micro-measurement results for AspectJ.

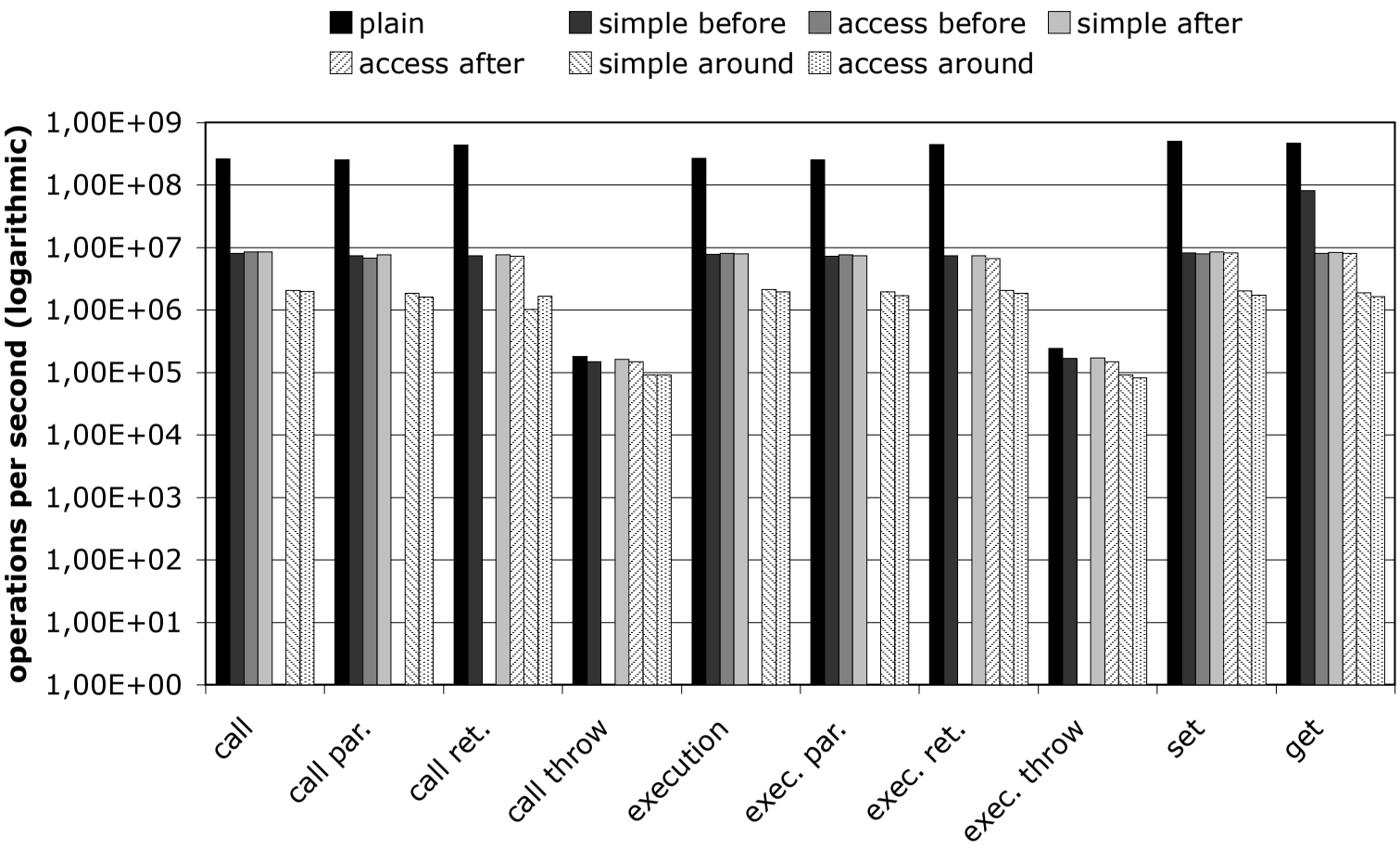

Figure A.2.: Micro-measurement results for CaesarJ.

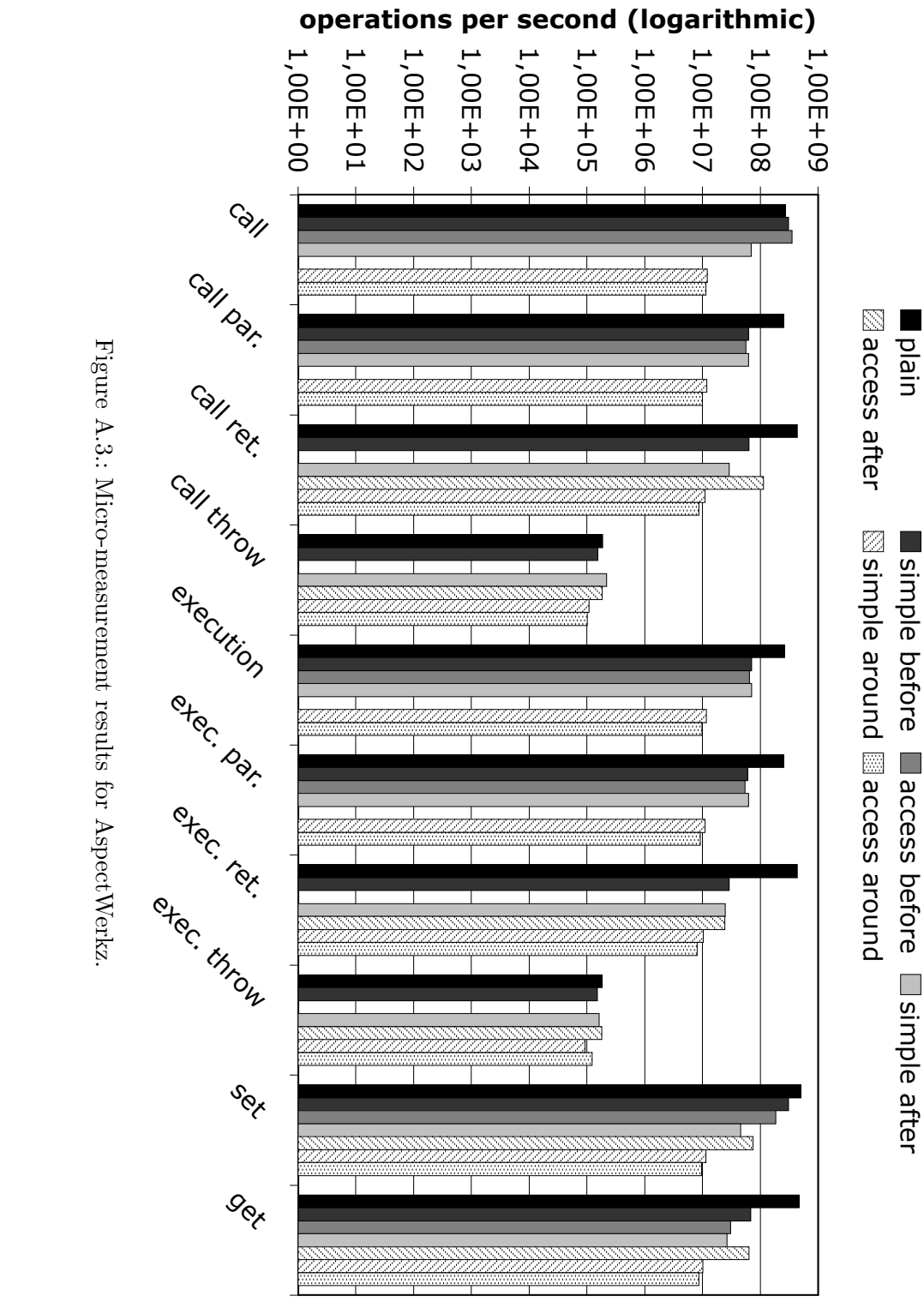

Figure A.3.: Micro-measurement results for AspectWerkz.

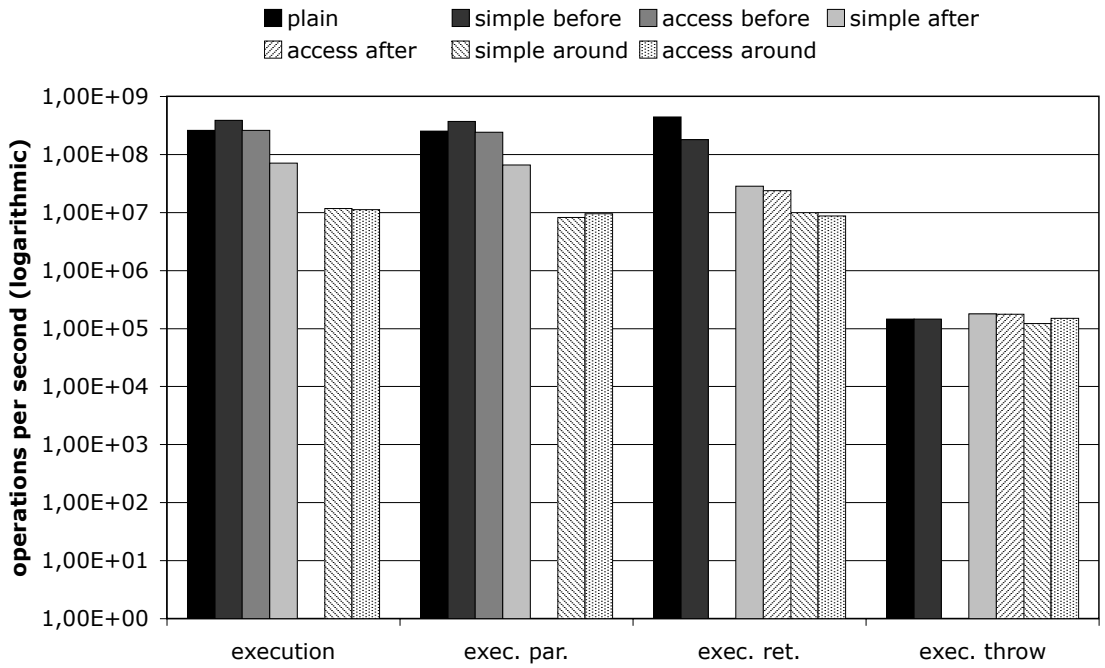

Figure A.4.: Micro-measurement results for JAsCo.

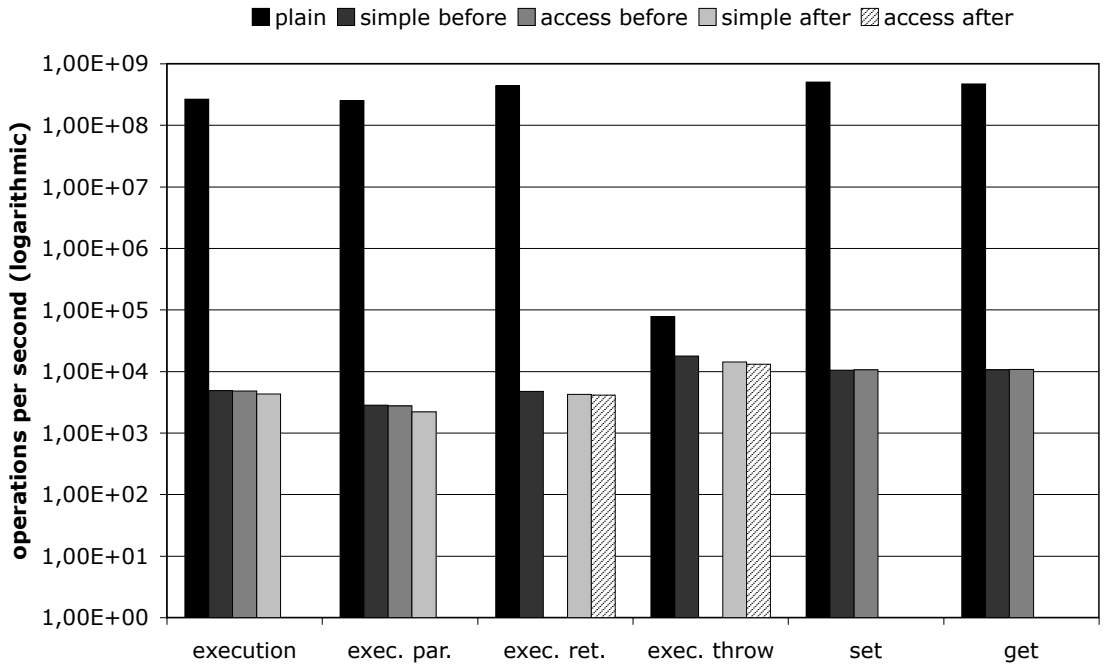

Figure A.5.: Micro-measurement results for PROSE.

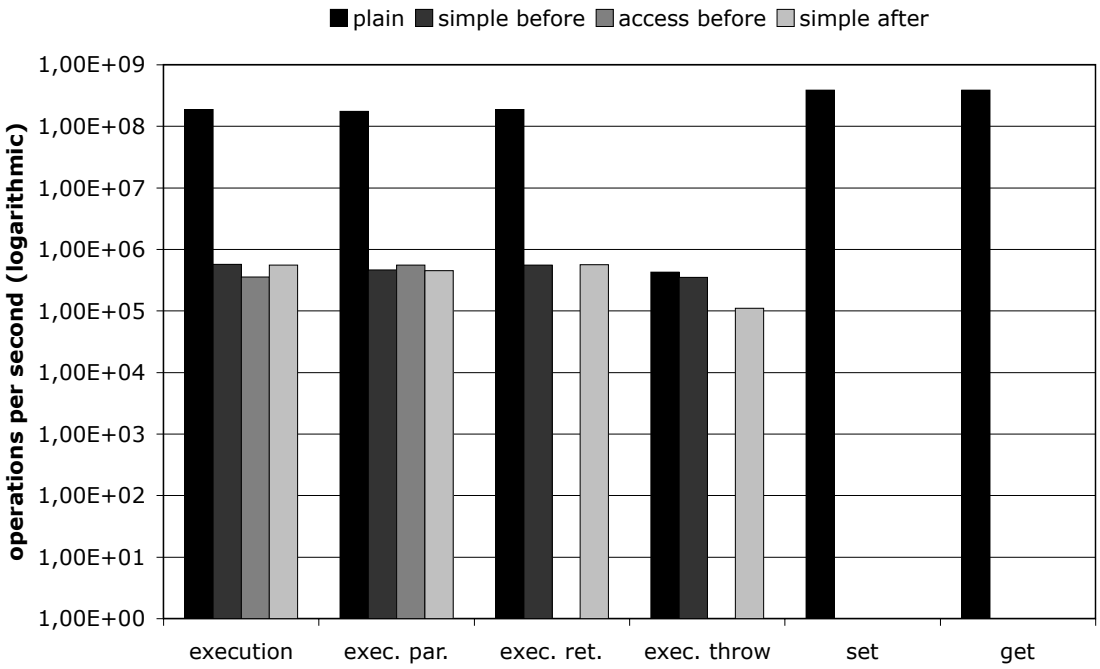

Figure A.6.: Micro-measurement results for PROSE/Jikes.

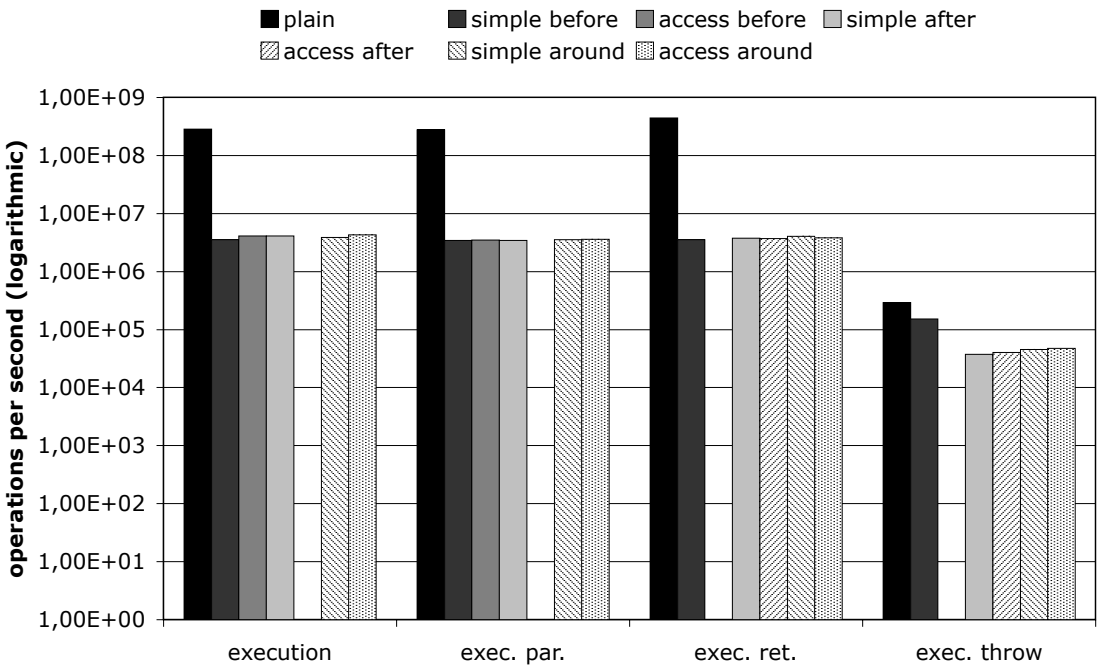

Figure A.7.: Micro-measurement results for Spring AOP.

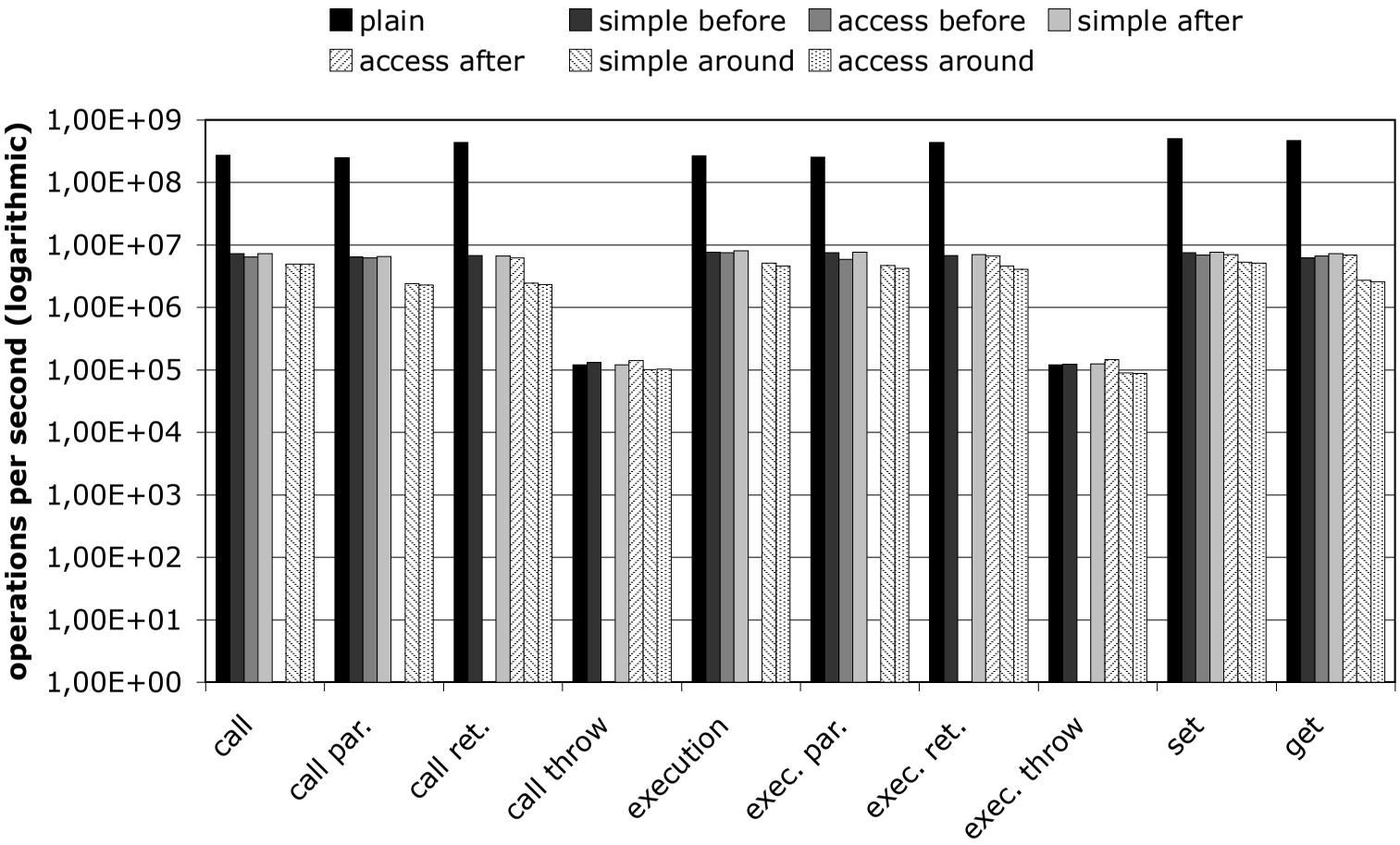

Figure A.8.: Micro-measurement results for Reflex.

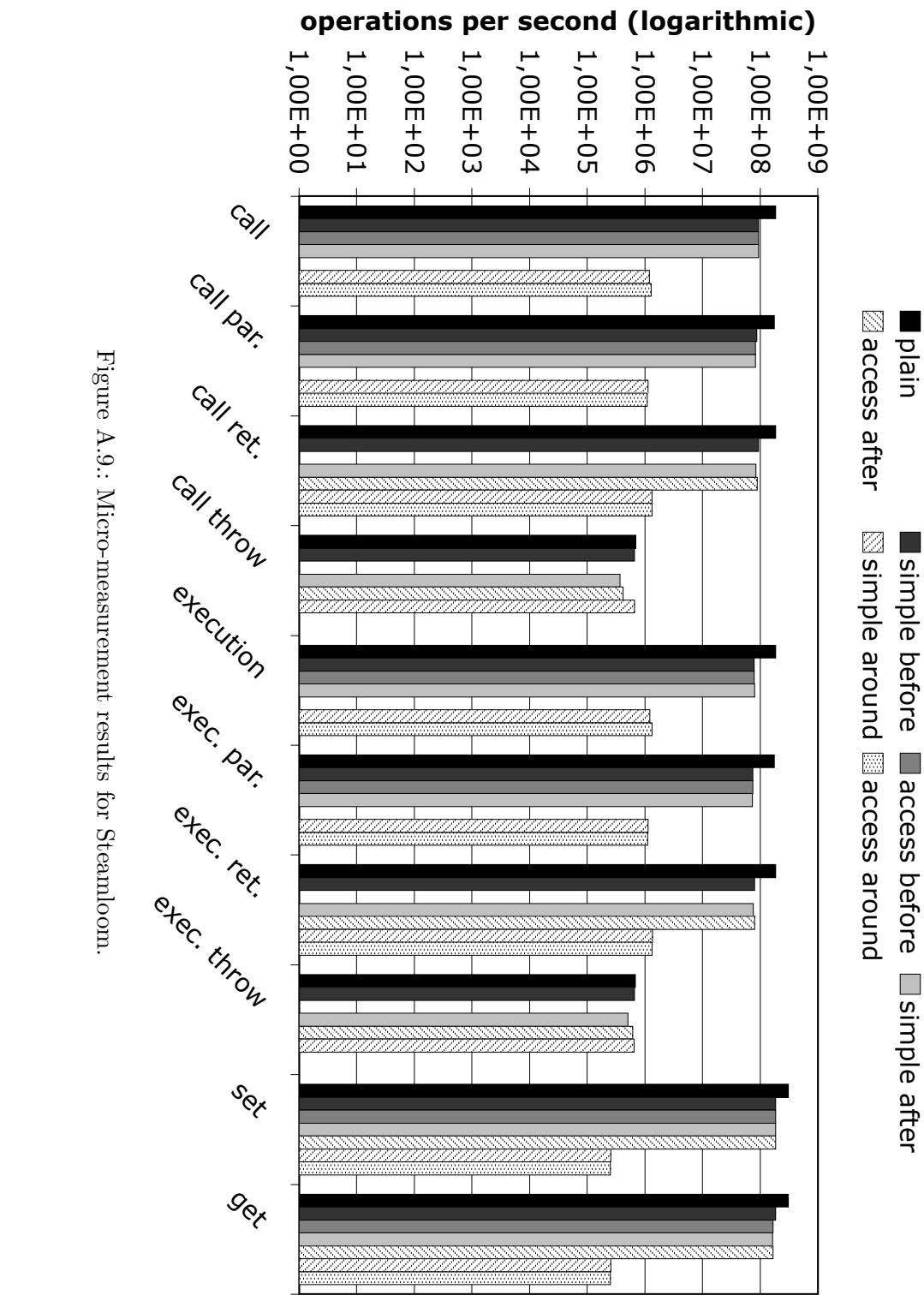

Figure A.9.: Micro-measurement results for Steamloom.

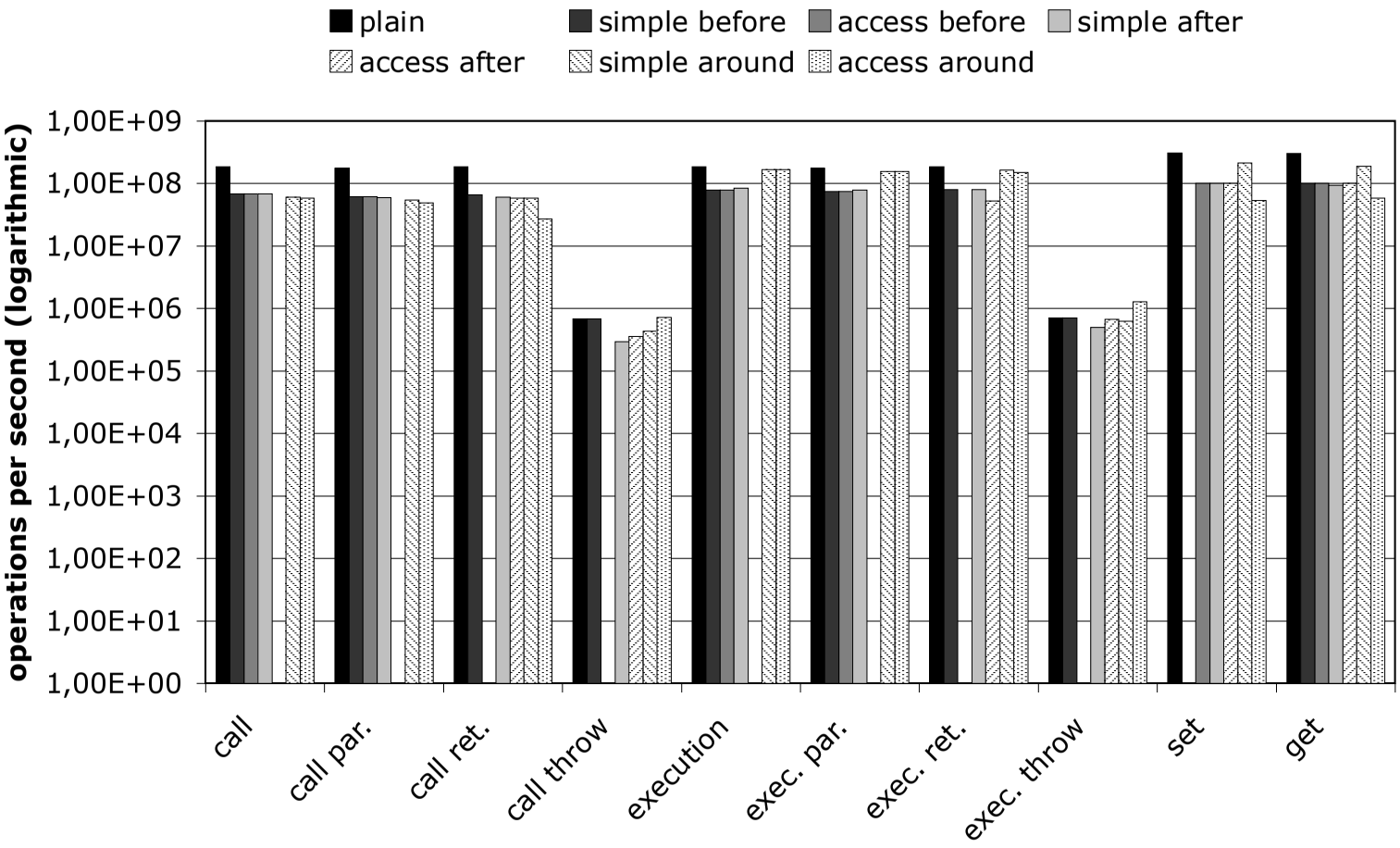

Figure A.10.: Micro-measurement results for AspectJ 1.2 on Jikes.

Bibliography

[1] abc (AspectBench Compiler) Home Page. `http://aspectbench.org/`.

[2] AspectJ Development Tools (Subproject of Eclipse) Home Page. `http://www.eclipse.org/ajdt/`.

[3] C. Allan et al. Adding Trace Matching with Free Variables to AspectJ. In *Proc. OOPSLA 2005*, pages 345–364. ACM Press, 2005.

[4] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*. ACM Press, 1999.

[5] B. Alpern, A. Cocchi, and D. Grove. Dynamic Type Checking in Jalapeño. In *Proceedings of JVM'01*. USENIX, 2001.

[6] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño—a compiler-supported java virtual machine for servers. ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS '99), May 1999.

[7] B. Alpern et al. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[8] B. Alpern et al. The Jikes Virtual Machine Research Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44(2):399–418, 2005.

[9] AOP Alliance Home Page. `http://aopalliance.sourceforge.net/`.

[10] AOSD-Europe Network of Excellence Project Home Page. `http://aosd-europe.net/`.

[11] Apostle Home Page. `http://www.cs.ubc.ca/labs/spl/projects/apostle/`.

[12] Arachne Home Page. `http://www.emn.fr/x-info/arachne/`.

[13] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *OOPSLA 2000 Proceedings*, pages 47–65. ACM Press, 2000.

[14] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM: The Controller's Analytical Model. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, 2000.

[15] M. Arnold, S. Fink, D. Grove, and P. F. Sweeney. Architecture and Policy for Adaptive Optimization in Virtual Machines. Technical Report RC23429 (W0411-125), IBM T. J. Watson Research Center, November 2004.

[16] M. Arnold, M. Hind, and B. G. Ryder. Online Feedback-Directed Optimization of Java. In *OOPSLA 2002 Proceedings*, pages 111–129. ACM Press, 2002.

[17] ASM Home Page. `http://asm.objectweb.org/`.

[18] AspectC++ Home Page. `http://www.aspectc.org/`.

[19] AspectJ Home Page. `http://www.eclipse.org/aspectj/`.

[20] AspectS Home Page. `http://www-ia.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/`.

[21] AspectWerkz Home Page. `http://aspectwerkz.codehaus.org/`.

[22] P. Avgustinov et al. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–128. ACM Press, 2005.

[23] AWBench AOP Benchmark Home Page. `http://docs.codehaus.org/display/AW/AOP+Benchmark`.

[24] BAT Home Page. `http://www.st.informatik.tu-darmstadt.de/BAT/`.

[25] BAT License. `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/BAT/License.html`.

[26] Byte Code Engineering Library (BCEL) Home Page. `http://jakarta.apache.org/bcel/`.

[27] BEA Home Page. `http://www.bea.com/`.

[28] BeanShell Project Home Page. `http://www.beanshell.org/`.

[29] A. Black, editor. *ECOOP 2005 - Object Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings*, volume 3586 of *LNCS*. Springer, 2005.

[30] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE*, pages 137–146, 2004.

[31] Blackdown Project Home Page. `http://www.blackdown.org/`.

[32] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proc. AOSD 2004*. ACM Press, 2004.

[33] J. Bonér. What Are the Key Issues for Commercial AOP Use: how Does AspectWerkz Address Them? In *Proc. AOSD 2004*, pages 5–6. ACM Press, 2004.

[34] J. Brichau and M. Haupt (editors). Survey of Aspect-Oriented Languages and Execution Models. `http://aosd-europe.net/documents/aspLang.pdf`, AOSD-Europe Network of Excellence, 2005.

[35] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[36] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Java Grande 1999 Proceedings*, pages 129–141. ACM Press, 1999.

[37] C. Bockisch and M. Haupt and M. Mezini and R. Mitschke. Evnelope-based Weaving for Faster Aspect Compilers. In *Proc. NetObjectDays 2005*. GI, 2005.

[38] CaesarJ Home Page. `http://caesarj.org/`.

[39] CARMA Home Page. `http://prog.vub.ac.be/~kgybels/Research/AOP.html`.

[40] S. Chiba. Load-Time Structural Reflection in Java. In Elisa Bertino, editor, *ECOOP 2000 – Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, 2000.

[41] R. Chitchyan and I. Sommerville. Comparing Dynamic AO Systems. In [61].

[42] M. Cilia, M. Haupt, M. Mezini, and A. P. Buchmann. The Convergence of AOP and Active Databases: Towards Reactive Middleware. In F. Pfenning and Y. Smaragdakis, editors, *Proc. GPCE 2003*, volume 2830 of *LNCS*, pages 169–188. Springer, 2003.

[43] A. Colyer. AOP@Work: Introducing AspectJ 5. `http://www-128.ibm.com/developerworks/java/library/j-aopwork8/`.

[44] Common Public License, version 1.0. `http://www.eclipse.org/legal/cpl-v10.html`.

[45] The DaCapo Benchmark Suite. `http://www-ali.cs.umass.edu/DaCapo/gcbm.html`.

[46] M. Dahm. Byte Code Engineering. In *JIT'99 Proceedings*. Springer, 1999.

[47] U. Dayal, A. P. Buchmann, and D. R. McCarthy. Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems*, volume 334 of *Lecture Notes In Computer Science*, pages 129–143. Springer, 1988.

[48] B. de Alwis. Aspects of Incremental Programming. Master's thesis, Department of Computer Science, University of British Columbia, April 2002.

[49] B. de Alwis and G. Kiczales. Apostle: A Simple Incremental Weaver for a Dynamic Aspect Language. Technical Report TR-2003-16, Department of Computer Science, University of British Columbia, Vancouver, Canada, 2003.

[50] M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Workshop on Engineering Complex Object-Oriented Systems for Evolution, Proceedings (at OOPSLA 2001)*, 2001.

[51] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An Expressive Aspect Language for System Applications with Arachne. In [150].

[52] R. Douence, O. Motelet, and M. Südholt. A Formal Definition of Crosscuts. pages 170–186. In [157].

[53] R. Douence and Mario Südholt. A Model and a Tool for Event-Based Aspect-Oriented Programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

[54] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the Dynamic Behaviour of AspectJ Programs. In *Proc. OOPSLA 2004*, 2004.

[55] Dynamic Proxy Classes (Java 1.3 feature). `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`.

[56] DynAOP Project Home Page. `https://dynaop.dev.java.net/`.

[57] EAOP Home Page. `http://www.emn.fr/x-info/eaop/`.

[58] M. Engel and B. Freisleben. Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects. pages 51–62. In [150].

[59] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[60] R. E. Filman, M. Haupt, and R. Hirschfeld (eds.). Proceedings of the 2005 Dynamic Aspects Workshop. Technical Report RIACS Technical Report No. 05.01, RIACS, 2005.

[61] R. E. Filman, M. Haupt, K. Mehner, and M. Mezini (eds.). Proceedings of the 2004 Dynamic Aspects Workshop. Technical Report RIACS Technical Report No. 04.01, RIACS, 2004.

[62] R. E. Filman and K. Havelund. Source-Code Instrumentation and Quantification of Events. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Workshop (at AOSD 2002)*, pages 45–49, 2002.

[63] S. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *Proc. CGO 2003*, pages 241–252. IEEE Computer Society, 2003.

[64] B. De Fraine, W. Vanderperren, D. Suvée, and J. Brichau. Jumping Aspects Revisited. In [60].

[65] FreeBSD Project Home Page. `http://www.freebsd.org/`.

[66] T. Fritz, M. Ségura-Devillechaise, M. Südholt, E. Wuchner, and J.-M. Menaud. An Application of Dynamic AOP to Medical Image Generation. In [60].

[67] E. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, School of Computer Science, McGill University, Montreal, Canada, 2002.

[68] E. Gagnon and L. Hendren. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, 2001.

[69] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[70] GNU C Compiler Project Home Page. `http://gcc.gnu.org/`.

[71] GNU Classpath Home Page. `http://www.gnu.org/software/classpath/classpath.html`.

[72] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[73] GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`.

[74] K. Gybels and J. Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *AOSD 2003 Proceedings*, pages 60–69. ACM Press, 2003.

[75] S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving. In *Proc. AOSD 2004*. ACM Press, 2004.

[76] M. Haupt and M. Mezini. Virtual Machine Support for Aspects with Advice Instance Tables. First French Workshop on Aspect-Oriented Programming (JFDLPA), Paris, France, Sep. 14th, 2004. `http://www.st.informatik.tu-darmstadt.de/database/publications/data/JFDLPA04.pdf?id=102`.

[77] M. Haupt and M. Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. In M. Weske and P. Liggesmeyer, editors, *Proc. Net.ObjectDays 2004*, volume 3263 of *LNCS*. Springer, 2004.

[78] M. Haupt and M. Mezini. Virtual Machine Support for Aspects with Advice Instance Tables. *L'Objet*, 11(3), 2005.

[79] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An Execution Layer for Aspect-Oriented Programming Languages. In *Proc. VEE 2005*. ACM Press, June 2005.

[80] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proc. Net.ObjectDays 2002*, 2002.

[81] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proc. AOSD 2004*. ACM Press, 2004.

[82] R. Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *LNCS*, pages 216–232. Springer, 2003.

[83] Iguana/J Home Page. `www.iguanaj.org/`.

[84] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In *Proc. OOPSLA 1997*, pages 318–326. ACM Press, 1997.

[85] J. Bonér and A. Vasseur and J. Dahlstedt. JRockit JVM Support for AOP, Part 1. `http://dev2dev.bea.com/pub/a/2005/08/jvm_aop_1.html?page=1`, 2005.

[86] J. Bonér and A. Vasseur and J. Dahlstedt. JRockit JVM Support for AOP, Part 2. `http://dev2dev.bea.com/pub/a/2005/08/jvm_aop_2.html?page=1`, 2005.

[87] J2EE: Java 2 Enterprise Edition Home Page. `http://java.sun.com/j2ee/`.

[88] JAC Home Page. `http://www.objectweb.org/`.

[89] JAsCo Home Page. `http://ssel.vub.ac.be/jasco/`.

[90] Java Native Interface Specification (Version 5.0). `http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html`.

[91] Javabeans specification. `http://java.sun.com/products/javabeans/docs/spec.html`.

[92] Javassist Home Page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/`.

[93] JBoss AOP Home Page. `http://www.jboss.com/products/aop`.

[94] Jeet Project Home Page. `http://jeet-aop.com/`.

[95] Jikes Compiler Home Page. `http://jikes.sourceforge.net/`.

[96] Jikes Bytecode Toolkit Home Page. `http://www.alphaworks.ibm.com/tech/jikesbt/`.

[97] The Jikes Research Virtual Machine. `http://jikesrvm.sourceforge.net/`.

[98] R. Johnson and J. Hoeller. *Expert One-on-One J2EE Development without EJB*. Wiley, 2004.

[99] R. Jones and R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[100] JRockit Home Page. `http://www.bea.com/products/weblogic/jrockit/index.shtml`.

[101] JVM Tools Interface Home Page. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`.

[102] K. Sakurai and H. Masuhara and N. Ubayashi and S. Matsuura and S. Komiya. Association Aspects. In *Proc. AOSD 2004*, pages 16–25. ACM Press, 2004.

[103] Kaffe Home Page. `http://www.kaffe.org/`.

[104] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[105] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[106] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[107] G. Kiczales and M. Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In [29].

[108] R. Laddad. *AspectJ in Action*. Manning, 2003.

[109] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[110] N. Loughran, N. Parlavantzas, M. Pinto, P. Sánchez, M. Webster, and A. Colyer. Survey of Aspect-Oriented Middleware. `http://aosd-europe.net/documents/middle.pdf`, AOSD-Europe Network of Excellence, 2005.

[111] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Proc. AOSD 2003*. ACM Press, 2003.

[112] H. Masuhara and G. Kiczales. Modeling Crosscutting Aspect-Oriented Mechanisms. In *Proc. ECOOP 2003*, 2003.

[113] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In G. Hedin, editor, *Proc. CC 2003*, volume 2622 of *LNCS*, pages 46–60. Springer, 2003.

[114] M. Mezini and K. Ostermann. Integrating Independent Components with On-Demand Remodularization. In *Proc. OOPSLA 2002*, pages 52–67. ACM Press, 2002.

[115] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136, 2004.

[116] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 2002.

[117] Nanning Aspects Home Page. `http://nanning.codehaus.org/`.

[118] NetBSD Project Home Page. `http://www.netbsd.org/`.

[119] A. Nicoara and G. Alonso. Dynamic AOP with PROSE. `http://www.iks.inf.ethz.ch/publications/publications/files/PROSE-ASMEA05.pdf`.

[120] O. Spinczyk and A. Gal and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*. ACM.

[121] S. Oaks. *Java Security*. O'Reilly, 2nd edition, 2001.

[122] ObjectTeams Home Page. `http://www.objectteams.org/`.

[123] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In [29].

[124] P. Avgustinov and others. abc: an Extensible AspectJ Compiler. pages 87–98. In [150].

[125] N. W. Paton. *Active Rules in Database Systems*. Springer, 1999.

[126] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, 2nd edition, 2002.

[127] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. pages 1–24. In [157].

[128] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In G. Kiczales, editor, *Proc. AOSD 2002*. ACM Press, 2002.

[129] A. Popovici, T. Gross, and G. Alonso. Just-in-Time Aspects. In *Proc. AOSD 2003*. ACM Press, 2003.

[130] PROSE Home Page. `http://prose.ethz.ch`.

[131] B. Redmond and V. Cahill. Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java. In *Proc. Workshop on Reflection and Meta-Level Architectures (co-located with ECOOP 2000)*, 2000.

[132] B. Redmond and V. Cahill. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. In B. Magnusson, editor, *ECOOP 2002: Object-Oriented Programming. Proceedings of the 16th European Conference, Málaga, Spain, June 2002.*, volume 2374 of *Lecture Notes in Computer Science (LNCS)*, pages 205–230. Springer, 2002.

[133] Reflex Home Page at INRIA. `http://www.emn.fr/x-info/reflex/`.

[134] Reflex Home Page at the University of Chile. `http://reflex.dcc.uchile.cl/`.

[135] L. Rodríguez, E. Tanter, and J. Noyé. Supporting Dynamic Crosscutting with Partial Behavioral Reflection: A Case Study. In *Proceedings of the 24th International Conference of the Chilean Computer Science Society (SCCC)*, pages 48–58. IEEE Computer Society, 2004.

[136] SableVM Home Page. `http://sablevm.org/`.

[137] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. Lawall. Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 110–119. ACM Press, 2003.

[138] SPECjbb2000 Home Page. `http://www.specbench.org/osg/jbb2000/`.

[139] SPECjvm98 Home Page. `http://www.spec.org/osg/jvm98/`.

[140] Spring Home Page. `http://www.springframework.org/`.

[141] Spring AOP (from the Spring reference documentation). `http://www.springframework.org/docs/reference/aop.html`.

[142] Squeak Home Page. `http://www.squeak.org/`.

[143] Sun Microsystems. The Java HotSpot(TM) Virtual Machine, v1.4.2, d2. A Technical White Paper. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf`, September 2002.

[144] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented Approach Tailored for Component Based Software Development. In *Proc. AOSD 2003*, pages 21–29, 2003.

[145] E. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, Ecole des Mines de Nantes, University of Chile, 2004.

[146] E. Tanter, N. M. N. Bouraqadi-Saâdani, and J. Noyé. Reflex – Towards an Open Reflective Extension of Java. pages 25–43. In [157].

[147] E. Tanter and J. Noyé. Versatile Kernels for Aspect-Oriented Programming. Technical Report RR No. 5275, INRIA, 2004.

[148] E. Tanter and J. Noyé. A Versatile Kernel for Multi-Language AOP. In R. Glück and M. Lowry, editors, *Proc. GPCE 2005*, volume 3676 of *LNCS*. Springer, 2005.

[149] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In *Proc. OOPSLA 2003*. ACM Press, 2003.

[150] P. Tarr, editor. *Aspect-Oriented Software Development. Proceedings of the 4th International Conference on Aspect-Oriented Software Development*. ACM Press, 2005.

[151] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 107–119. ACM Press, 1999.

[152] W. Vanderperren and D. Suvee. Optimizing JAsCo Dynamic AOP through HotSwap and Jutta. In [61].

[153] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. `http://ssel.vub.ac.be/jasco/media/sc2005.pdf`.

[154] J. Whaley. Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[155] Y. Yanagisawa, S. Chiba, and K. Kourai. A Source-level Kernel Profiler based on Dynamic Aspect Orientation. In [60].

[156] F. Yellin. Low Level Security in Java. In *Fourth International World Wide Web Conference Proceedings*, volume 1 of *World Wide Web Journal*. O'Reilly, 1995.

[157] A. Yonezawa and S. Matsuoka, editors. *Metalevel Architectures and Separation of Crosscutting Concerns, Third International Conference, REFLECTION 2001, Kyoto, Japan, September 25-28, 2001, Proceedings*, volume 2192 of *LNCS*. Springer, 2001.

Curriculum Vitae

The curriculum vitae of this work's author, Michael Haupt, is as follows:

June 4, 1975	born in Olpe, Germany
1981–1985	Paul-Gerhardt-Grundschule, Attendorn, Germany
1985–1994	St.-Ursula-Gymnasium, Attendorn, Germany graduated with German Abitur
1994–2000	University of Siegen, Germany studies in „Technische Informatik“ (computer science as an engineering discipline) graduated with diploma
2001–2006	Darmstadt University of Technology Software Technology Group graduated with doctoral degree